COMPUTATION OF DYNAMIC SLICES OF ASPECT-ORIENTED PROGRAMS

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology
in
Computer Science and Engineering

By

Madhusmita Sahu

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY
ROURKELA

2007

COMPUTATION OF DYNAMIC SLICES OF ASPECT-ORIENTED PROGRAMS

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology
in
Computer Science and Engineering

By

Madhusmita Sahu

Under the Guidance of

Prof. D. P. Mohapatra

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY
ROURKELA

2007

National Institute of Technology
Rourkela

CERTIFICATE

This is to certify that the thesis titled **"Computation of Dynamic Slices of Aspect-Oriented Programs"**, submitted by **Madhusmita Sahu** in partial fulfillment of the requirements for the award of Master of Technology Degree in **Computer Science and Engineering** with specialization in **"Computer Science"** at the National Institute of Technology, Rourkela (Deemed University) is an authentic work carried out by her under my supervision and guidance.

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other university/institute for the award of any degree.

Prof. D. P. Mohapatra
Date: **Dept. of Comp. Sc. & Engg.,**
National Institute of Technology
Rourkela-769008

Acknowledgement

I wish to express my heartiest thanks to all who extended their unlimited help to me during my research work and its subsequent documentation.

I wish to express my sincere gratitude to my guide, Dr. D. P. Mohapatra, for his kind and able guidance for the completion of this research work. His consistent support and intellectual guidance made me energize and innovate new ideas.

I am grateful to Dr. S. K. Jena, Head of the Department, Computer Science Engineering, NIT Rourkela for his support during my work.

I am thankful to all professors and lecturers and members of the department of Computer Science and Engineering, NIT, Rourkela for their generous help in various ways for the completion of the research work.

I would like to thank my classmates at NIT, Rourkela for their generous help in various ways which resulted in the completion of the Documentation.

Madhusmita Sahu

Contents

Abstract

This thesis presents our work concerning computation of dynamic slicing of aspect-oriented programs.

Program slicing is a decomposition technique which extracts program elements related to a particular computation from a program. A *program slice* consists of those parts of a program that may directly or indirectly affect the values computed at some program point of interest, referred to as a *slicing criterion*.

A program slice can be static or dynamic. *Static slice* contains all the statements that may affect the slicing criterion for every possible inputs to the program. *Dynamic slice* contains only those statements that actually affect the slicing criterion for a particular input to the program.

Aspect-Oriented Programming is a new programming technique proposed for cleanly modularizing the cross-cutting structure of concerns. An *aspect* is an *area of concern* that cuts across the structure of a program. The main idea behind Aspect-Oriented Programming (AOP) is to allow a program to be constructed by describing each concern separately.

AspectJ is an aspect-oriented extension to the Java programming language. AspectJ adds new concepts and associated constructs called join points, pointcuts, advices, introductions, aspects to Java.

We first store the statements executed for a particular execution in an execution trace file. Next, we develop a dependence-based representation called *Dynamic Aspect-Oriented Dependence Graph* (DADG) as the intermediate program representation. The DADG is an arc-classified digraph which represents various dynamic dependences between the statements of an aspect-oriented program for a particular execution.

Then, we present an efficient dynamic slicing technique for aspect-oriented programs using DADG. Taking any vertex as the starting point, our algorithm performs a graph traversal on the DADG using breadth-first graph traversal or depth-first graph traversal. Then, the traversed vertices are mapped to the original program to compute the dynamic slices.

We have shown that our proposed algorithm efficiently calculates dynamic slices. The space complexity of the algorithm is $O(S)$. The run-time complexity of the algorithm is $O(S^2)$. We have also shown that our dynamic slicing algorithm computes correct dynamic slices.

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Program Slicing

Categories of Program Slicing

Issues in Program Slicing

Motivation

Objectives

Organization of the Thesis

1.1 Program Slicing

Program Slicing is a technique for aiding debugging and program comprehension by reducing complexity. Program slicing is a decomposition technique which extracts program elements related to a particular computation from a program [66]. The concept of a program slice was first introduced by Mark Weiser [43, 44]. A *program slice* consists of those parts of a program that may directly or indirectly affect the values computed at some program point of interest, referred to as a *slicing criterion* . Thus a program slice consists of all statements in a program P that may affect the value of variable v at some point p [7]. The task to compute program slices is called *program slicing*. Finding all statements in a program that directly or indirectly affect the value of a variable occurrence is referred to as *Program Slicing* [25]. The statements selected constitute a *slice* of the program with respect to the variable occurrence. A program is sliced with respect to a *slicing criterion* . A slicing criterion is a tuple $< s, v >$, where s is a program point (statement) of interest and v is a variable used or defined at s.

Program slicing transforms a large program into a smaller one that contains only statements relevant to the computation of a given function. Various slightly different notions of program slices have been proposed. There has also been a proliferation of the number of methods to compute slices. The main reason for this proliferation of slicing techniques is that different applications require different properties of slices. Weiser [44] defined a program slice S as a *reduced, executable program* obtained from a program P by removing statements, such that S replicates part of the behavior of P. The program slicing technique originally introduced by Weiser [44] is now called *static backward slicing*. It is *static* in the sense that the slice is independent of the input values to the program. It is *backward* because the control flow of the program is considered in reverse while constructing the slice. Another common definition of a slice is a subset of the statements and control predicates of the program which directly or indirectly affect the values computed at the slicing criterion , but which do not necessarily constitute an executable program. Program slicing has many applications in software engineering activities including program understanding, debugging, testing, maintenance and model checking, program comprehension etc. [46, 52, 60, 66].

Aspect-oriented programming (AOP) is a new programming paradigm that was proposed by Gregor Kiczales et al. [22] for modularizing the crosscutting structure of different concerns such as exeception handling, synchronization, logging, security, resource sharing, user interface. When such cross-cutting concerns are expressed using standard language constructs, it produces poorly structured code since these concerns are tangled with the basic functionality of the code. This increases the system complexity and makes maintenance considerably more difficult.

AOP [1, 2, 32] attempts to solve this problem by allowing the programmer to develop

cross-cutting concerns as full stand-alone modules called aspects. The main idea behind AOP is to allow a program to be constructed by describing each concern separately.

Aspect-oriented programming languages present unique opportunities and problems for program analysis schemes. For example, to perform program slicing on aspect-oriented software, specific aspect-oriented features such as join-point, advice, aspect, introduction must be handled appropriately. Although these features provide great strengths to model the crosscutting concerns in an aspect-oriented program, they introduce difficulties to analyze the program.

Researchers have developed many representations for procedural and object-oriented programs [18, 31, 37, 46, 48, 49, 57, 69], but very few work has been carried out for representation of aspect-oriented programs [33, 35, 59]. Due to the specific features of aspect-oriented programming languages, existing slicing algorithms for procedural or object-oriented programming languages cannot be applied directly to aspect-oriented programs. Zhao [33] was the first to consider the aspect-oriented features in his work. He developed the aspect-oriented system dependence graph (ASDG) to represent aspect-oriented programs and used the two-pass slicing algorithm of Larsen and Harrold [37] to compute static slices.

A major goal of any dynamic slicing technique is *efficiency* since results are normally used during interactive applications such as program debugging. Effciency is an especially important concern in slicing aspect-oriented programs, since the size of practical aspect-oriented programs is often very large. The response time of an inefficient dynamic slicer may be unacceptably large for such programs. In all slicing techniques, the source code is first analyzed to produce a graph representation called an *intermediate program representation.* Then the intermediate program representation is analyzed by using an algorithm to compute the slice. So, the effciency of a slicing technique depends on how suitably the program is represented by an intermediate representation and how much effcient the slicing algorithm is.

1.2 Categories of Program Slicing

Several categories of program slicing as well as methods to compute them are found in literature. The main reason for the existence of so many categories of slicing is the fact that different applications require different types of slices.

- **Static Slicing and Dynamic Slicing**

  Static Slicing technique uses static analysis to derive slices. That is, the source code of the program is analyzed and the slices are computed for all possible input values. Therefore conservative assumptions are made which often lead to relatively larger slices.

A static slice contains all the statements that may affect the slicing criterion for every possible inputs to the program. Thus, a static slice may contain statements that might not be executed during an actual run of a program. Static program slicing isolates all possible threads computing a particular variable. A static slice of a program P with respect to a slicing criterion $< s, V >$ is the set of all the statements of P that might affect the values of the variables in V at the program point in s.

Dynamic Slicing makes use of the information about a particular execution of a program. A dynamic slice with respect to slicing criterion $< s, V >$, for a particular execution, contains those statements that actually affect the slicing criterion in the particular execution. Therefore dynamic slices are usually smaller than static slices. Dynamic slicing preserves the program's behavior for a specific program input rather than that for the set of all inputs. Dynamic slicing isolates the unique thread computing the variable for the given inputs. Dynamic slice contains only those statements that actually affect the slicing criterion. In other words, dynamic slicing techniques compute precise slices.

Consider the C++ example program given in Fig 1.1. The static slice with respect to the slicing criterion $< 11, sum >$ is the set of statements {4, 5, 6, 8, 9}. Consider a particular execution of the program with the input value i=15. The dynamic slice with respect to the slicing criterion$< 11, sum >$ for the particular execution of the program is 5.

- **Backward Slicing and Forward Slicing**

  A *backward slice* contains all parts of the program that might directly or indirectly affect the slicing criterion. Thus a static backward slice provides the answer to the question: "which statements affect the slicing criterion?". It is computed by working backwards from the point of interest, finding all statements that can affect the specified variables at the point of interest and discarding the other statements. In the computation of dynamic slice, after the execution trace of the program is first recorded, the dynamic slice algorithm traces backwards the execution trace to derive data and control dependencies that are then used to compute the dynamic slice.

  A *forward slice* with respect to a slicing criterion $< s, V >$ contains all parts of the program that might be affected by the variables in V used or defined at the program point s. A forward slice provides the answer to the question: "which statements will

```
1       main()
2       {
3         int i,sum;
4         cin>>i;
5         sum=0;
6         while(i<=10)
7         {
8           sum=sum+i;
9           ++i;
10        }
11        cout<<sum;
12        cout<<i;
13      }
```

Figure 1.1: An Example Program

be affected by the slicing criterion?". Forward slicing works forward from the point of interest finding those statements that can be affected by changes to the specified variables at the point of interest. For programs with very long executions, the forward slicing is used. Here, dynamic slice is computed during program execution and execution trace is not recorded.

- **Intra-Procedural Slicing and Inter-Procedural Slicing**

  Intra-procedural slicing computes slices within a single procedure. Calls to other procedures are either not handled at all or handled conservatively.

  If the program consists of more than one procedure, *inter-procedural slicing* can be used to derive slices that span multiple procedures.

- **Other Slicing Categories**

  It is possible to combine the features of static slicing with the features of dynamic slicing. This new form of slicing is called *hybrid slicing* [53, 54]. Hybrid slicing is an approach for refining static slices using dynamic information.

  There are variants of slicing in between the two extremes of static and dynamic, where some but not all properties of the initial state are known. These are known as *conditioned slices* or *constrained slices*. Traditional slicing methods are all based on statement deletion. In a recently reported form of slicing called *amorphous slicing* [39], slices are not necessarily produced by deleting statements and may not necessarily even be made from components of the original program being sliced. The slice is

computed based on the *semantics* of the program. Recently, another form of slicing called *modular monadic slicing* has been developed where slices are computed based on the modular monadic semantics of the program analyzed. This method computes slices directly on abstract syntax of the program without constructing intermediate representations such as dependence graphs.

1.3 Issues in Program Slicing

In this section, we discuss some of the major issues in dynamic slicing of aspect-oriented programs.

- **Intermediate Representation:** In order to slice an aspect-oriented program, first the program should be represented by a suitable intermediate representation. This intermediate representation should correctly represent the aspect-oriented features such as join points, pointcuts, advices, inter-type declarations. We have developed a suitable intermediate representation in Chapter 5, which correctly represents these aspect-oriented features.

- **Memory Requirement:** The memory requirement for both the intermediate representation and the dynamic slicing algorithm should be as small as possible. Otherwise, the stored data will run out of memory due to the large sizes of aspect-oriented programs. We have shown that our intermediate representation and dynamic slicing algorithm are more space efficient i.e., it requires less memory space.

- **Time Requirement:** The time requirement for any dynamic slicing algorithm should also be as small as possible as the algorithm will be generally used in interactive applications such as debugging. Otherwise, the response time will be too large. We have shown that our dynamic slicing algorithm is more time efficient i.e., it requires less amount of time to compute dynamic slices.

- **Correctness:** The dynamic slicing algorithm should compute correct dynamic slices with respect to any given slicing criterion. A slice is said to be correct if it contains all the statements that affect the slicing criterion. We prove that the proposed dynamic slicing algorithm computes correct dynamic slices with respect to any given slicing criterion.

- **Scalability:** The dynamic slicing algorithms should be developed in such a way that the algorithms can easily be extended to handle large scale programs as the sizes of practical aspect-oriented programs are very large. Our dynamic slicing algorithm can easily be extended to handle large and complex programs.

1.4 Motivation

It has been observed that smaller slices are more useful for different applications. So, the major aim of any slicing technique is to realize as small a slice with respect to a slicing criterion as possible. Much of the literature on program slicing is concerned with the improvement of the algorithms for slicing in terms of the reduced size of the slice and the improvement of the efficiency of the slicing algorithm.

Aspect-oriented programming is a new concept. The aspect-oriented programs are quite large. It is difficult to debug and test these programs. Program slicing techniques have been found to be useful in program understanding, debugging, testing, software maintenance, reverse engineering etc. Dynamic program slicing is used in interactive applications such as debugging and testing of programs. This requires the development of efficient dynamic slicing techniques and suitable intermediate representations for aspect-oriented programs. The reports on slicing of aspect-oriented programs are very few and also these are less efficient. Thus, there is a need to develop suitable intermediate representations and efficient algorithms for dynamic slicing of aspect-oriented programs.

In the next section, the major goals of this thesis are identified.

1.5 Objectives

The main objective of our research work is to develop efficient dynamic slicing algorithms. For this purpose, we identify the following goals.

- Computation of dynamic slices of aspect-oriented programs as fast as possible. For this, we plan to develop:
  - suitable intermediate representation for aspect-oriented programs which can be used for slicing algorithm.
  - development of dynamic slicing algorithm for aspect-oriented programs using the proposed intermediate representation.
- Efficient computation of dynamic slices to interactively confine bugs in an aspect-oriented program because dynamic slicing is used for debugging purposes. The technique is to be space and time efficient.
- Implementation of the proposed algorithm to verify its *correctness* experimentally.
- Evaluation of the performance of the algorithm in terms of space and time.

1.6 Organization of the Thesis

The rest of this thesis is organized into chapters as follows.

Chapter 2 provides background concepts used in the rest of the thesis. We describe some graph-theoretic concepts and various definitions which will be used later in our algorithms. We present some intermediate program representation concepts which are used in slicing techniques. Then, we briefly discuss the concepts of *precision* and *correctness* of a dynamic slice. Finally, we present various applications of program slicing.

Chapter 3 presents a brief review of the related work relevant to our contribution. First, we discuss the work carried out on dynamic slicing of object-oriented programs. Then, we describe the work carried out on slicing of aspect-oriented programs.

Chapter 4 discusses the various concepts and features of aspect-oriented programming. First, we present the basic concepts of aspect-oriented programming. Then, we briefly describe various aspect-oriented languages. Finally, we present the features of AspectJ [1, 2, 3, 20, 21, 33, 35], which is a popular aspect-oriented language.

Chapter 5 presents our dynamic slicing algorithms for simple aspect-oriented programs. We introduce some basic concepts and definitions which will be used in our algorithms. First, we develop an intermediate program representation for aspect-oriented programs and then, present our dynamic slicing algorithm. Finally, we discuss the *correctness* and *complexity* of our algorithm.

Chapter 6 provides a brief discussion on the implementation of our algorithm. First, we present a brief introduction to Lex and YACC and then, we describe about our slicer.

Chapter 7 concludes the thesis with a summary of our contributions. Also, we briefly discuss the possible future extensions to our work.

Chapter 2

BACKGROUND

Definitions

Program Representation

Precision and Correctness of a Slice

Applications of Program Slicing

The area of program slicing has been enriched over the last two decades by contributions from several researchers. The technique of program slicing has been extended to handle unstructured and multi-procedure programs, structured as well as object-oriented and aspect-oriented programs.

This chapter provides a gist of the background used in the rest of the thesis. Section 2.1 contains some basic definitions. Section 2.2 describes some intermediate program representations concepts which are commonly used in slicing techniques. Section 2.4 features some important applications of program slicing.

2.1 Definitions

Definition 1 (Directed Graph):A directed graph G is a pair (N, E) where N is a finite non-empty set of elements called nodes and $E \subseteq N \times N$ is a set of directed edges between the nodes.

Let $G = (N, E)$ be graph. If (x, y) is an edge of G, then x is called a *predecessor* of y and y is called a *successor* of x. G contains two special nodes, $n_{initial}$, which has no predecessors, and n_{final}, which has no successors. A *directed path* (or *path*) from a node x_1 to a node x_k in a graph $G = (N, E)$ is a sequence of nodes $(x_1, x_2, \ldots, x_k)$ such that $(x_i, x_{i+1}) \in E$ for every $i, 1 \leq i \leq k-1$. Furthermore, there is a path from $n_{initial}$ to every node in G and a path from n_{final} to every node in G^{-1}, the inverse graph of G. A path that has actually been executed for some input is referred to as an *executable trace*.

Definition 2 (Flow Graph): A *flow graph* is a quadruple, $(N, E, Start, Stop)$ where (N, E) is a graph, $Start \in N$ is a distinguished node of in-degree 0 called the *start node*, $Stop \in N$ is a distinguished node of out-degree 0 called the *stop node*. There is a path from *Start* to every other node in the graph, and there is a path from every other node in the graph to *Stop*.

Definition 3 (Dominance): If x and y are two nodes in a control flow graph, then x dominates y iff every path from *Start* to y passes through x. y post-dominates x iff every path from x to *Stop* passes through y.

Let x and y be nodes in a flow graph G. Node x is said to be immediate post-dominator of node y iff x is a post-dominator of y, $x \neq y$ and each post-dominator $z \neq x$ of y post-dominates x. The post-dominator tree of a flow graph G is the tree that consists of the nodes of G, has the root *Stop*, and has an edge (x, y) iff x is the immediate post-dominator of y.

Consider the flow graph of the example program of Figure 2.1, which is given in Figure 2.2. In the flow graph, each of the nodes 4, 5 and 6 dominates 7. Node 8 does not dominate node 10. Node 10 post-dominates each of the nodes 4, 5, 6, 7, 8 and 9. Node 9 post-dominate

```
1        main()
2        {
3          int x,y,prod;
4          cin>>x;
5          cin>>y;
6          prod=1;
7          while(x<5)
8          {
9            prod=prod*y;
10          ++x;
11         }
12         cout<<prod;
13         prod=y;
14         cout<<prod;
15      }
```

Figure 2.1: An Example Program

node 8. Node 9 post-dominates none of the nodes 4, 5, 6, 7, 10, 11 and 12. Node 6 is the immediate post-dominator of node 5. Node 10 is the immediate post-dominator of node 7.

2.2 Program Representation

Various types of program representation schemes exist which include high level source code, pseudocode, a set of machine instructions in a computer's memory, a flow chart and others. Different representations may be required to facilitate human readability, annotation for verifiability and transformation for running a program on platform such as multiprocessors and distributed computers, etc. In the context of program slicing, program representations are used to support efficient automation of slicing.

A slice for any given slicing criterion can be determined manually for a simple program with less complexity. But, with increasing size and complexity of the programs, automatic slice computation is essential. Current automated slicing techniques require that the information available in a source code form of the program to be sliced be transformed into some mathematical representation during the slicing process.

In the following, a few basic concepts associated with intermediate program representations are presented.

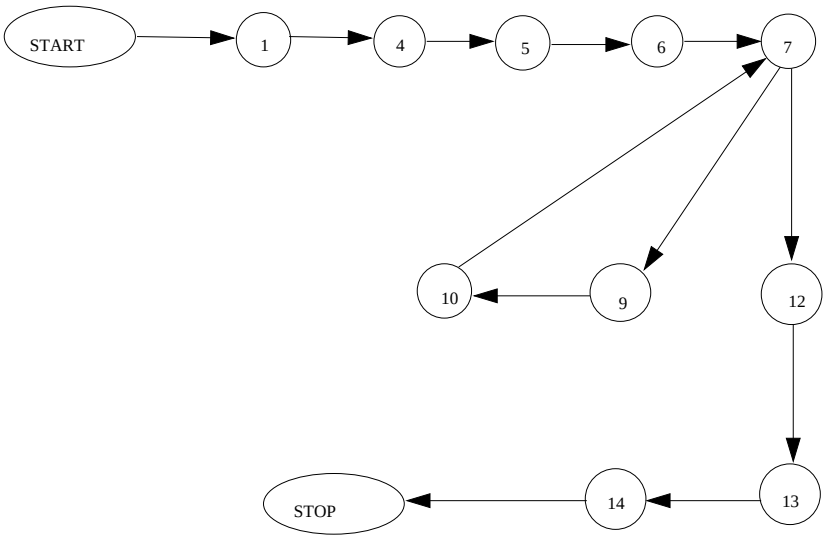

Figure 2.2: The CFG of example program given in Figure 2.1

2.2.1 Control Flow Graph (CFG)

A *control flow graph (CFG)* is an intermediate representation for programs that is useful for data flow analysis and for many optimizing code transformations such as common subexpression elimination, copy propagation and loop invariant code motion. It is a graph for a program P in which each node is associated with a statement from P and the edges represent the flow of control in P.

Definition 4 (Control Flow Graph): Let the set N represent the set of statements of a program P. The *control flow graph* of the program P is the flow graph $G = (N_1, E, \mathit{Start}, \mathit{Stop})$ where $N_1 = N \cup \mathit{Start}, \mathit{Stop}$. An edge $(m, n) \in E$ indicates the possible flow of control from the node m to the node n.

The existence of an edge (x, y) in the CFG means that the control *must* transfer from x to y during program execution. Figure 2.2 represents the CFG of example program given in Figure 2.1. The CFG of a program P models the branching structures of the program, and it can be built while parsing the source code using algorithms that have linear time complexity in the size of the program.

2.2.2 Data Dependence Graph

The CFG of a program represents the flow of control through the program. The flow of data through a program is often more useful in program analysis. Data flow describes the flow of the values of variables from the points of their definitions to the points where their values are used.

Definition 5 (Data Dependence): Let G be the CFG of a program P. A node n is said to be data dependent on a node m if there exists a variable *var* of the program P such that the following hold:

(i) the node m defines *var*

(ii) the node n uses *var*

(iii) there exists a directed path from m to n along which there is no intervening definition of *var*.

Consider the example program given in Figure 2.1 and its CFG in Figure 2.2. The node 8 has data dependence on each of the nodes 5, 6 and 8. The node 11 has data dependence on the node 5. Node 11 has data dependence on none of the nodes 6 and 8.

Definition 6 (Data Dependence Graph): The *data dependence graph* of a program P is the graph $G = (N, E)$, where each node $n \in N$ represents a statement of the program P and $(x, y) \in E$ iff x is data dependent on y.

2.2.3 Control Dependence Graph

The *control dependence graph* is used as an intermediate representation to represent the relations between program variables arising due to control flow.

Definition 7 (Control Dependence): Let G be the control flow graph of a program P. Let x and y be nodes in G. Node y is *control dependent* on node x if the following hold:

(i) x is a test node

(ii) there exists a directed path Q from x to y such that none of the internal nodes (nodes excluding x and y) is a jump node

(iii) y post-dominates every $z \neq x$ in Q

(iv) y does not post-dominate x.

Let x and y be two nodes in the CFG G of a program P. If y is control dependent on x, then x must have multiple successors in G. Conversely, if x has multiple successors, then at least one of its successors must be control dependent on it.

Consider the example program given in Figure 2.1 and its CFG in Figure 2.2. Each of the nodes 8 and 9 is control dependent on the node 7. The node 7 has two successor nodes 8 and 10 and the node 8 has control dependence on node 7.

Definition 8 (Control Dependence Graph): The *Control dependence graph* of a program P is the graph $G = (N, E)$, where each node $n \in N$ represents a statement of the program P and $(x, y) \in E$ iff x is control dependent on y.

2.2.4 Program Dependence Graph (PDG)

Program dependence graphs combine control dependences and data dependences into a common framework. The nodes of a program dependence graph represent statements and predicate expressions of the program. Each node of the graph has references to the nodes that it is control dependent on and to the nodes that define its operands. The set of all dependences induce a partial ordering on the statements and predicates in the program that must be followed in order to preserve the semantics of the original program. Since both the essential control relationships and the essential data relationships are present in the program dependence graph, a single traversal of these dependences is sufficient to perform many optimizations.

Definition 9 (Program Dependence Graph): The *program dependence graph* G of a program P is a graph $G = (N, E)$, where each node $n \in N$ represents a statement of the program P. The graph contains two kinds of directed edges: *control dependence edges* and *data dependence edges*. A control (or data) dependence edge (n, m) indicates that n is control (or data) dependent on m.

The PDG of a program P is the union of a pair of graphs: the *data dependence graph* of P and the *control dependence graph* of P. The vertices of PDG for program P, denoted by G_P, represent the assignment statements and control predicates that occur in program P. In addition, G_P includes three other categories of vertices:

(i) There is a distinguished vertex called the *entry vertex*.

(ii) For each variable x for which there is a path in the standard control-flow graph for P on which x is used before being defined, there is a vertex called the *initial definition of* x. This vertex represents an assignment to x from the initial state. The vertex is labeled "x:= InitialState(x)".

(iii) For each variable x named in P's end statement, there is a vertex called the *final use of* x. It represents an access to the final value of x computed by P, and is labeled "FinalUse(x)".

The edges of G_P represent *dependences* among program components. An edge represents either a *control dependence* or a *data dependence*. Control dependence edges are labeled either **true** or **false**, and the source of a control dependence edge is always the entry vertex or a predicate vertex. A control dependence edge from vertex v_1 to vertex v_2, denoted by $v_1 \longrightarrow_c v_2$, means that, during execution, whenever the predicate represented by v_1 is evaluated and its value matches the label on the edge to v_2, then, the program component represented by v_2 will eventually be executed if the program terminates. The program

dependence graph G_P contains a *control dependence edge* from vertex v_1 to vertex v_2 of G_P iff one of the following holds:

(1) v_1 is the entry vertex and v_2 represents a component of P that is not nested within any loop or conditional; these edges are labeled **true**.

(2) v_1 represents a control predicate and v_2 represents a component of P immediately nested within the loop or conditional whose predicate is represented by v_l. If v_1 is the predicate of a while-loop, the edge $v_1 \longrightarrow_c v_2$ is labeled *true*; if v_1 is the predicate of a conditional statement, the edge $v_1 \longrightarrow_c v_2$ is labeled true or false according to whether v_2 occurs in the *then* branch or the *else* branch, respectively.

A data dependence edge from vertex v_1 to vertex v_2 means that the programs computation might be changed if the relative order of the components represented by v_1 and v_2 were reversed. The program dependence graphs contain two kinds of data dependence edges, representing *flow dependences* and *def-order dependences*. The data dependence edges of a program dependence graph are computed using data-flow analysis.

A program dependence graph contains a flow dependence edge from vertex v_1 to vertex v_2 iff all of the following hold:

(1) v_1 is a vertex that defines variable x.

(2) v_2 is a vertex that uses x.

(3) Control can reach v_2 after v_1 via an execution path along which there is no intervening definition of x. That is, there is a path in the standard control flow graph for the program by which the definition of x at v_1 reaches the use of x at v_2. (Initial definitions of variables are considered to occur at the beginning of the control-flow graph; final uses of variables are considered to occur at the end of the control-flow graph.)

A flow dependence that exists from vertex v_1 to vertex v_2 is denoted by $v_1 \longrightarrow_f v_2$.

Flow dependences can be further classified as *loop carried* or *loop independent*. A flow dependence $v_1 \longrightarrow_f v_2$ is carried by loop L, denoted by $v_1 \longrightarrow_{lc(L)} v_2$, if in addition to (1), (2), and (3) above, the following also hold:

(4) There is an execution path that both satisfies the conditions of (3) above and includes a backedge to the predicate of loop L.

(5) Both v_1 and v_2 are enclosed in loop L.

A flow dependence $v_1 \longrightarrow_f v_2$ is loop-independent, denoted by $v_1 \longrightarrow_{li} v_2$, if in addition to (l), (2), and (3) above, there is an execution path that satisfies (3) above and includes no

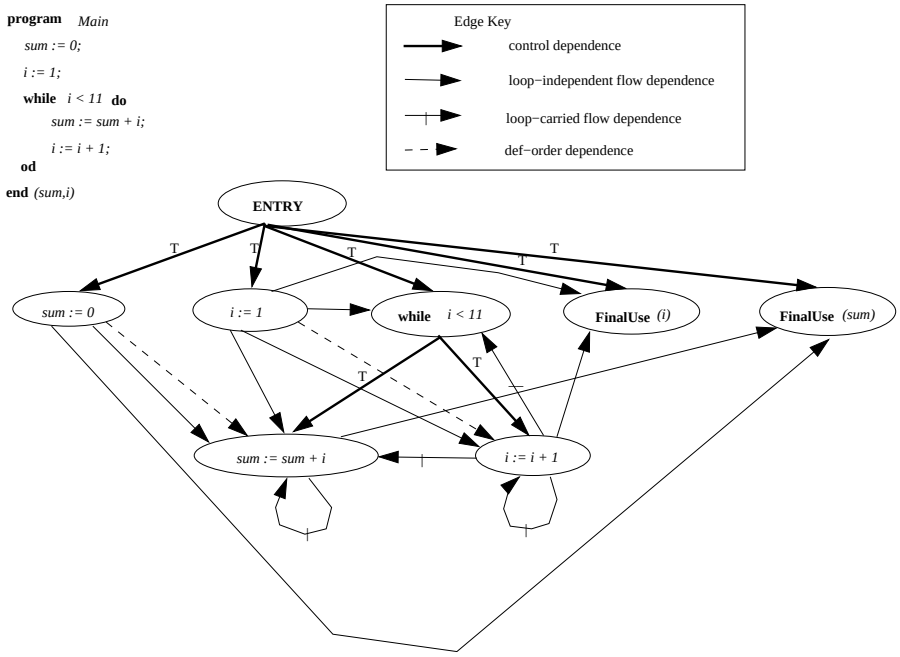

Figure 2.3: An example program and its program dependence graph

backedge to the predicate of a loop that encloses both v_1 and v_2. It is possible to have both $v_1 \longrightarrow_{lc(L)} v_2$ and $v_1 \longrightarrow_{li} v_2$.

A program dependence graph contains a def-order dependence edge from vertex v_1 to vertex v_2 iff all of the following hold:

(1) v_1 and v_2 both define the same variable.

(2) v_1 and v_2 are in the same branch of any conditional statement that encloses both of them.

(3) There exists a program component v_3 such that $v_1 \longrightarrow_f v_3$ and $v_2 \longrightarrow_f v_3$.

(4) v_1 occurs to the left of v_2 in the programs abstract syntax tree.

A def-order dependence from v_1 to v_2 with "witness" v_3 is denoted by $v_1 \longrightarrow_{do(v_3)} v_2$.

Note that a program dependence graph is a multigraph (i.e., it may have more than one edge of a given kind between two vertices). When there is more than one loop-carried flow dependence edge between two vertices, each is labeled by a different loop that carries the dependence. When there is more than one def-order edge between two vertices, each is labeled by a vertex that is flow-dependent on both the definition that occurs at the edges source and the definition that occurs at the edges target. Figure 2.3 shows an example program and its program dependence graph.

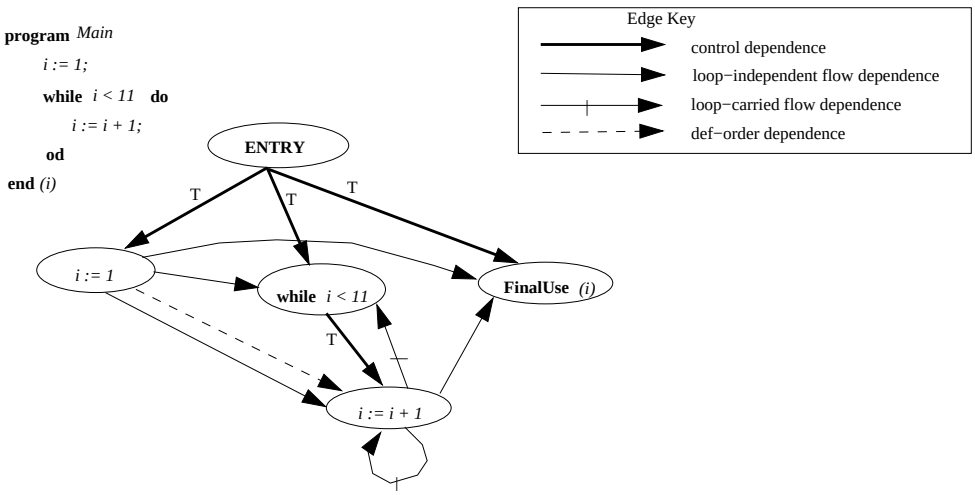

Figure 2.4: The graph and the corresponding program that result from slicing the program dependence graph from Figure 2.3 with respect to the final-use vertex for i.

The boldface arrows represent control dependence edges; solid arrows represent loop-independent flow dependence edges; solid arrows with a hash mark represent loop-carried flow dependence edges; dashed arrows represent def-order dependence edges.

For vertex s of program dependence graph G, the slice of G with respect to s is a graph containing all vertices on which s has a transitive flow or control dependence (i.e., all vertices that can reach s via flow and/or control edges). Figure 2.4 shows the graph that results from taking a slice of the program dependence graph from Figure 2.3 with respect to the final-use vertex for i, together with the one program to which it corresponds. The significance of an intraprocedural slice is that it captures a portion of a programs behavior in the sense that, for any initial state on which the program halts, the program and the slice compute the same sequence of values for each element of the slice. A program point may be

(1) an assignment statement,

(2) a control predicate, or

(3) a final use of a variable in an end statement.

Because a statement or control predicate may be reached repeatedly in a program by "computing the same sequence of values for each element of the slice", we mean:

(1) for any assignment statement the same *sequence* of values are assigned to the target variable;

(2) for the predicate the same sequence of Boolean values are produced; and

(3) for each final use the same value for the variable is produced.

2.2.5 System Dependence Graph (SDG)

The PDG is well suited for *intraprocedural slicing*. To solve the *interprocedural slicing*, *system dependence graph* (SDG) is used. A *system dependence graph* contains one *procedure dependence graph* for each procedure. A *procedure dependence graph* represents a procedure as a graph in which vertices are statements or predicate expressions. A *system dependence graph* extends *program dependence graph* to incorporate collections of procedures rather than just monolithic programs *i.e.* it represents multi-procedure programs.

The SDG models a language with the following properties:

(1) A complete system consists of a single (main) program and a collection of auxiliary procedures.

(2) Procedures end with **return** statements instead of **end** statements. A **return** statement does not include a list of variables.

(3) Parameters are passed by value-result.

A system dependence graph includes a *program dependence graph*, which represents the systems main program, *procedure dependence graphs*, which represent the systems auxiliary procedures, and some additional edges. These additional edges are of two sorts:

(1) edges that represent direct dependences between a call site and the called procedure, and

(2) edges that represent transitive dependences due to calls.

An SDG is made up of a collection of procedure dependence graphs connected by interprocedural *control dependence* edges and *flow-dependence* edges. *Flow-dependence* or *data-dependence* edges represent flow of data between statements or expressions; *control dependence* edges represent control conditions on which the execution of a statement or expression depends. Each procedure dependence graph contains an *entry* vertex that represents entry into the procedure. Extending the definiton of dependence graphs to handle procedure calls requires representing the passing of values between procedures. This model of parameter passing is represented in procedure dependence graphs through the use of five new kinds of vertices. A call site is represented using a *call-site* vertex; information transfer is represented using four kinds of *parameter* vertices. On the calling side, information transfer is represented by a set of vertices called *actual-in* and *actual-out* vertices. These vertices, which are control dependent on the call-site vertex, represent assignment statements that copy the values of the actual parameters to the call temporaries and from the return temporaries, respectively. Similarly, information transfer in the called procedure is represented by

a set of vertices called *formal-in* and *formal-out* vertices. These vertices, which are control dependent on the procedures entry vertex, represent assignment statements that copy the values of the formal parameters from the call temporaries and to the return temporaries, respectively.

Using this model, data dependences between procedures are limited to dependences from actual-in vertices to formal-in vertices and from formal-out vertices to actual-out vertices. Transitive dependence edges, called *summary* edges, are added from actual-in vertices to actual-out vertices to represent transitive flow dependences due to called procedures. A summary edge is added if a path of control flow and summary edge exists in the called procedure from the corresponding formal-in vertex to the corresponding formal-out vertex. The addition of a summary edge in procedure Q may complete a path from a formal-in vertex to a formal-out vertex in Q's PDG, which in turn may enable the addition of further summary edges in procedures that call Q. Connecting procedure dependence graphs to form a system dependence graph involves the addition of three new kinds of edges:

(1) a *call* edge is added from each call-site vertex to the corresponding procedure-entry vertex;

(2) a *parameter-in* edge is added from each actual-in vertex at a call site to the corresponding formal-in vertex in the called procedure;

(3) a *parameter-out* edge is added from each formal-out vertex in the called procedure to the corresponding actual-out vertex at the call site.

Figure 2.5 shows an example system and the corresponding SDG.

An interprocedural slice with respect to vertex s is computed in two phases. Both Phases 1 and 2 operate on the system dependence graph traversing edges to find the set of vertices that can reach a given set of vertices along certain kinds of edges. The traversal in Phase 1 follows flow edges, control edges, call edges and parameter-in edges, but does not follow def-order edges or parameter-out edges. The traversal in Phase 2 follows flow-edges, control edges and parameter-out edges, but does not follow def-order edges, call edges or parameter-in edges. Summary edges permit moving across a call site without having to descend into the called procedure. The two phases can be characterized as follows:

Phase 1. Phase 1 identifies vertices that can reach s, and are either in P itself or in a procedure that calls P (either directly or transitively). Because parameter-out edges are not followed, the traversal in Phase 1 does not "descend" into procedures called by P. The effects of such procedures are not ignored, however; the presence of *transitive flow dependence* edges from actual-in to actual-out vertices (subordinate-characteristic-graph edges) permits the discovery of vertices that can reach s only through a procedure call, although the graph

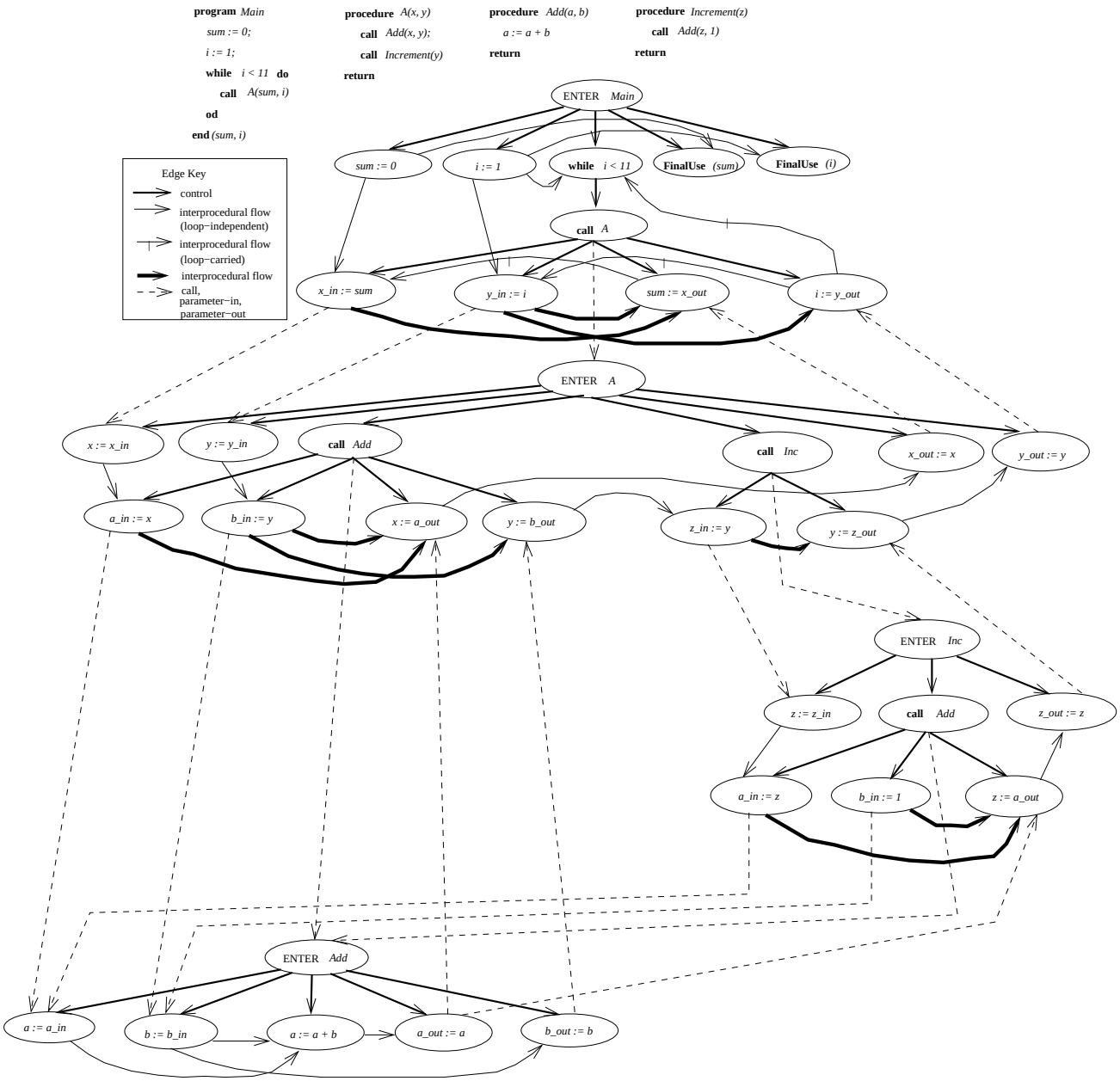

Figure 2.5: Example system and corresponding system dependence graph. Control dependences, shown unlabeled, are represented using medium-bold arrows; intraprocedural flow dependences are represented using arcs; transitive interprocedural flow dependences (corresponding to subordinate characteristic graph edges) are represented using heavy, bold arcs; call edges, parameter-in edges, and parameter-out edges (which connect program and procedure dependence graphs together) are represented using dashed arrows.

traversal does not actually descend into the called procedure.

Phase 2. Phase 2 identifies vertices that can reach s from procedures (transitively) called by P or from procedures called by procedures that (transitively) call P. Because call edges and parameter-in edges are not followed, the traversal in Phase 2 does not "ascend" into calling procedures; the transitive flow dependence edges from actual-in to actual-out vertices make such "ascents" unnecessary.

The result of an interprocedural slice consists of the sets of vertices encountered during by phase 1 and phase 2 and the set of edges induced by this vertex set.

Figure 2.6 and Figure 2.7 illustrate the two phases of the interprocedural slicing algorithm. Figure 2.6 shows the vertices of the example system dependence graph that are marked during Phase 1 of the interprocedural slicing algorithm when the system is sliced with respect to the formal-out vertex for parameter z in procedure *Increment*. Edges "traversed" during Phase 1 are also included in Figure 2.6. Figure 2.7 adds (in boldface) the vertices that are marked and the edges that are traversed during Phase 2 of the slice.

The result of an interprocedural slice consists of the sets of vertices identified by Phase 1 and Phase 2 and the set of edges induced by this vertex set. Figure 2.8 shows the completed example slice (excluding def-order edges.)

2.3 Precision and Correctness of a Slice

If P be a program and S be a static slice of P with respect to a slicing criterion C, then the reduced program S is itself an executable program and its behavior with respect to the slicing criterion C must be identical to the original program's behavior [43]. A slice S of P with respect to a slicing criterion C is *statement-minimal* if no other slice of P with respect to the slicing criterion has fewer statements than S.

A static slice can also be defined as follows: a slice S of a program P with respect to a slicing criterion C is a subset of the program statements which directly or indirectly affect the slicing criterion. It is to be noted that such a slice need not be executable.

It is reasonable to define the precision of a dynamic slice. A dynamic slice is said to be *precise* if it includes only those statements that actually affect the slicing criterion for the given execution. However, it is very difficult to determine the *preciseness* of a given slice since the determination of a precise slice is an undecidable problem.

Consider the example program given in Figure 2.9. In this example, the statement 6 uses the variable x and it has dependence on the statements 2 and 4 since x is defined at the statements 2 and 4.

It is to be noted that a precise dynamic slice need not be a *statement-minimal* slice.

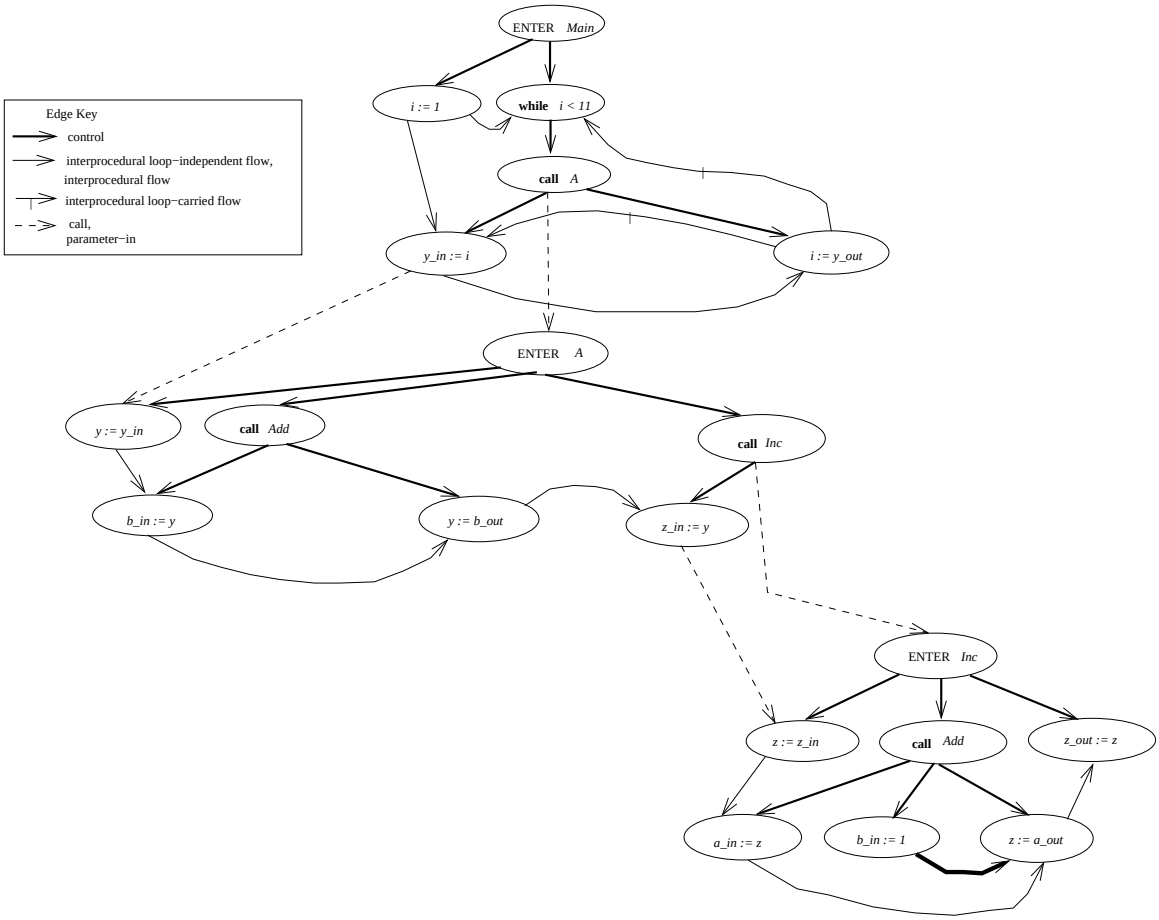

Figure 2.6: The example programs system dependence graph is sliced with respect to the formal-out vertex for parameter z in procedure *Increment*. The vertices marked by Phase 1 of the slicing algorithm as well as the edges traversed during this phase are shown above.

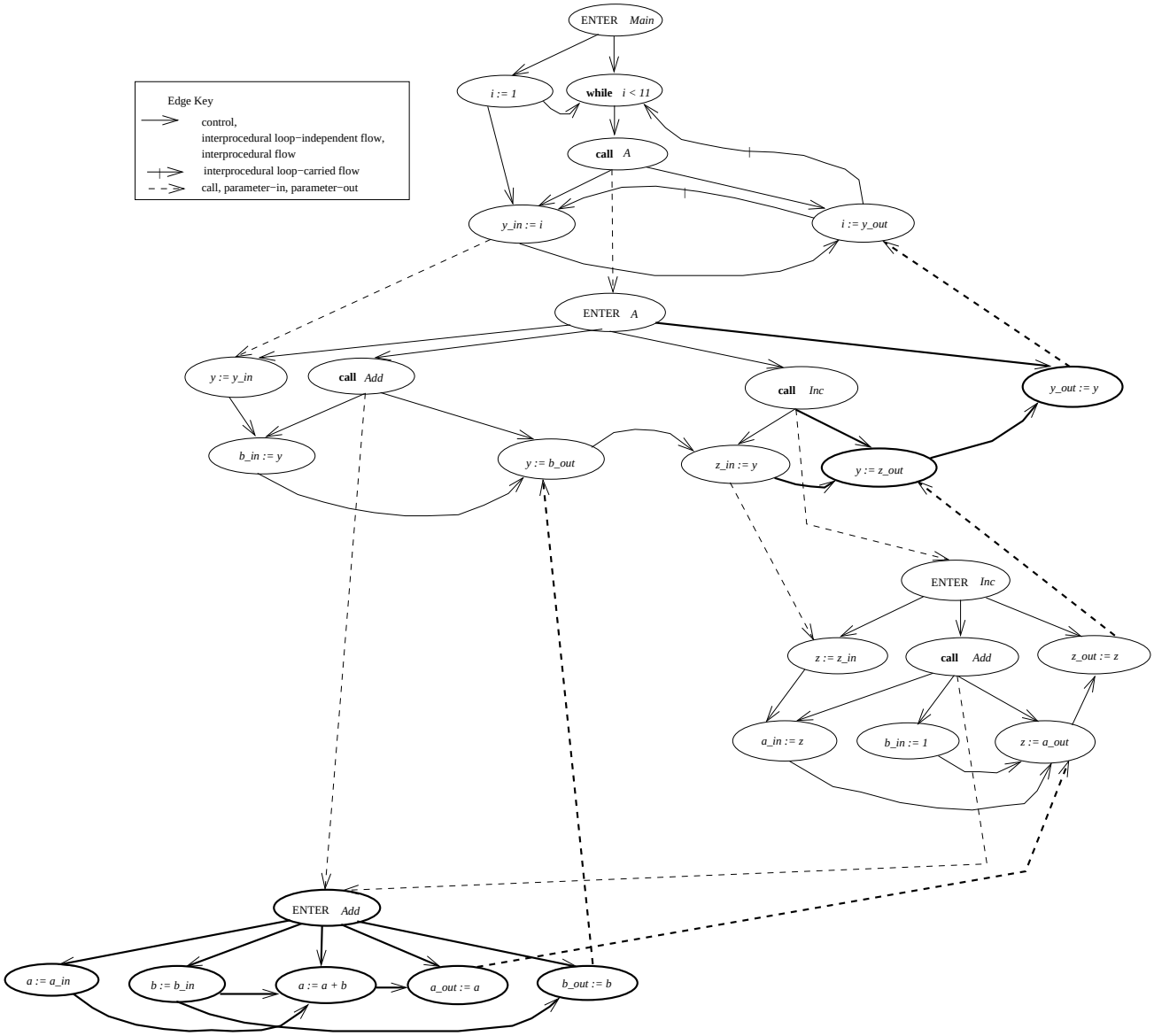

Figure 2.7: The example programs system dependence graph is sliced with respect to the formal-out vertex for parameter z in procedure *Increment*. The vertices marked by Phase 2 of the slicing algorithm as well as the edges traversed during this phase are shown above in boldface.

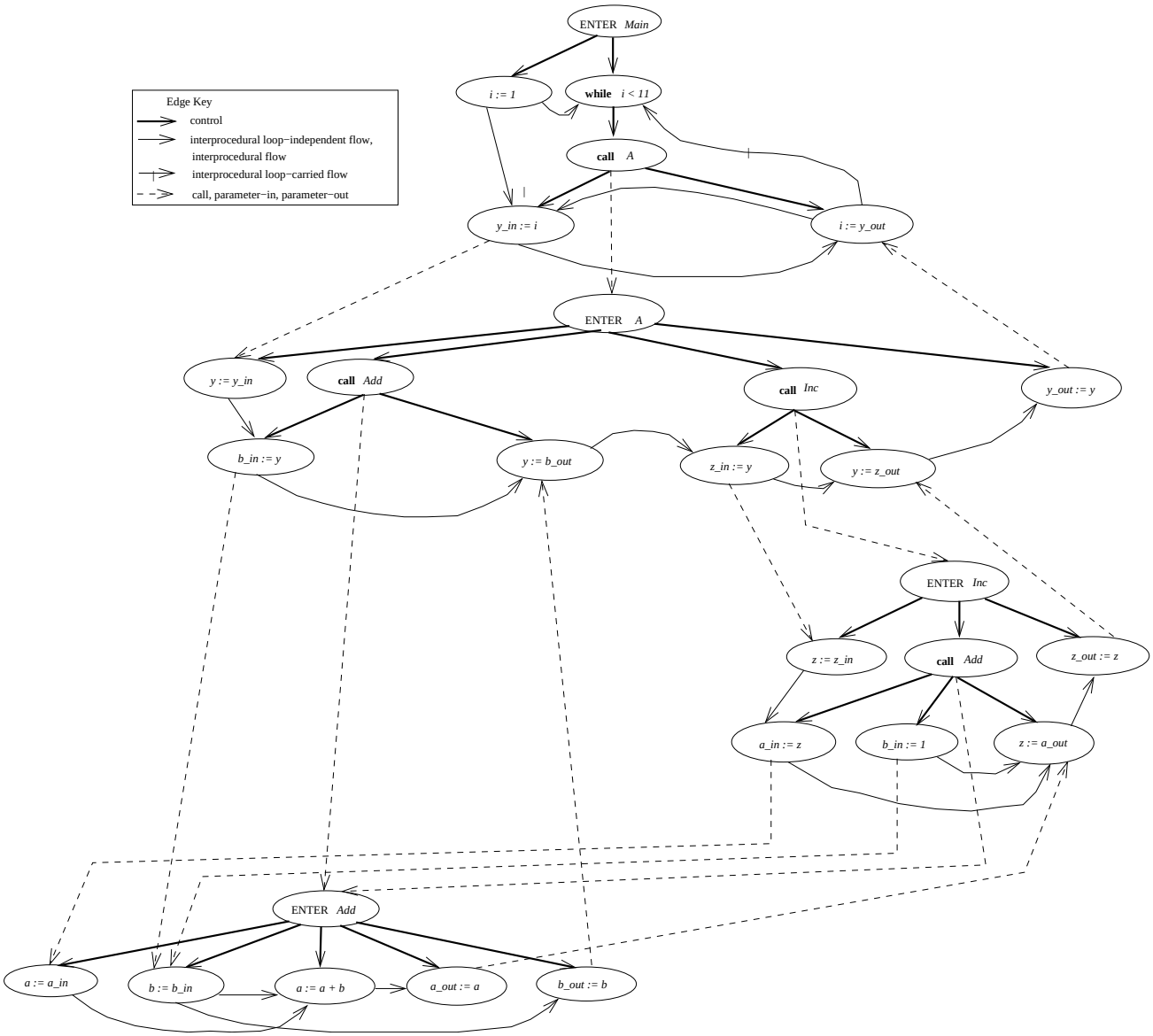

Figure 2.8: The complete slice (excluding def-order edges) of the example programs system dependence graph sliced with respect to the formal-out vertex for parameter z in procedure *Increment*.

```
1    i = 1;
2    x = 5;
3    while ( i < = 10 )
      {
4       x = x + 5;
5       i = i + 1;
      }
6    write ( x );
```

Figure 2.9: An example program

For illustration, consider the example program given in Figure 2.9. The final value of x at statement 6 is dependent only on statement 4. So, the *statement-minimal* slice with respect to the slicing criterion $< 6, x >$ is $\{4\}$. In the existing program slicing frameworks, the precise slice for the slicing criterion is $\{2, 4\}$ as these two statements are the reaching definitions of the variable x at statement 6.

A *correct* slice contains all the statements that affect the slicing criterion. On the other hand, an *incorrect* slice fails to contain some statements that affect the slicing criterion. It can be noted that the whole program is always a correct slice of any slicing criterion. A correct slice is said to be *imprecise* if it contains at least one statement that does not affect the slicing criterion.

2.4 Applications of Program Slicing

This section describes the use of program slicing techniques in various applications. The program slicing technique was originally developed to realize automated static code decomposition tools. The primary objective of those tools was to aid program debugging. From this modest beginning, the use of program slicing techniques has now ramified into a powerful set of tools for use in such diverse applications as program understanding, program verification, automated computation of several software engineering metrics, software maintenance and testing, functional cohesion, dead code elimination, reverse engineering, parallelization of sequential programs, software portability, reusable component generation, compiler optimization, program integration, showing differences between programs, software quality assurance, software fault-injection etc. In the following, some of these applications of program slicing are discussed briefly.

2.4.1 Differencing

Program differencing is the task of analyzing an old and a new version of a program in order to determine the set of program components of the new version that represent *syntactic* and *semantic* changes. Such information is useful because only the program components reflecting changed behavior need to be tested. The key issue in program differencing consists of partitioning the components of the old and new version in a way that two components are in the same partition only if they have equivalent behaviors. Algorithms for finding *textual* differences between programs (or arbitrary files) are often insufficient. Program slicing can be used to identify semantic differences between two programs. There are two related programs differencing problems:

1. Find all the components of two programs that have different behavior.

2. Produce a program that captures the semantic differences between two programs.

For programs *old* and *new*, a straightforward solution to Problem 1 is obtained by comparing the backward slices of the vertices in *old* and *new*'s dependence graphs G_{old} and G_{new}. Components whose vertices in G_{new} and G_{old} have isomorphic slices have the same behavior in *old* and *new*; thus, the set of vertices from G_{new} for which there is no vertex in G_{old} with an isomorphic slice *safely approximates* the set of components *new* with changed behavior. This set is a *safe* as it is guaranteed to contain all the components with different behavior. It is (necessarily) an *approximation* because the exact differencing problem is unsolvable.

The vertices in G_{new} with different behavior than in G_{old} the set of *affected points*. The complexity of the straight forward solution for finding affected points is cubic in the size of G_{new} (slice isomorphism can be determined in linear time). This set can be effciently computed in linear time using a single forward slice starting from the set of *directly affected points*: those vertices of G_{new} with different incoming dependence edges than in G_{old}.

A solution to the second differencing problem is obtained by taking the backward slice with respect to the set of affected points. For programs with procedure and procedure calls, two modifications are necessary: First, the inter-procedural slicing techniques are required to be used to ensure that the resulting program is executable. Second, this solution is overly pessimistic: consider a component c in procedure P that is called from two call-sites c_1 and c_2. If c is identified as an affected point by a forward slice that enters P through c_1 then, assuming there is not other connection, we will include c_1 but not c_2 in the program that captures the differences. However, the backward slice with respect to c would include both c_1 and c_2.

2.4.2 Debugging

Finding bugs in a program is always a difficult task. The process of finding a bug usually involves running the program over and over, learning more and narrowing down the search each time until the bug is finally located. Program slicing was discovered as an operation performed by programmers while debugging a piece of code. Programmers mentally slice a code while debugging it. Program debugging remains a main application area of slicing techniques even after several advancements to the slicing techniques. A tool that computes program slices is a valuable aid in debugging. It allows the programmer to focus attention on those statements that contribute to a fault. Also, highlighting a slice assists in uncovering faults caused by a statement that should be in a slice but is not.

Several kinds of slices are useful in debugging. Dynamic slicing is one variation of program slicing. It assists the programmmer in debugging. During debugging, a programmer normally has a test case which causes the program to fail. A dynamic slice is better suited to locate a bug exhibited on a particular execution of the program since it contains less of the program than a static slice.

Slicing is also useful in algorithmic debugging. In this process, starting from an external point of failure, the debugging algorithm localizes the bug to within a procedure by asking the programmer a series of questions. These questions relate to the expected behavior of a procedure. Program slicing allows one to ignore many statements in the process of localizing the bug.

Other variants of program slicing include *program dicing* and *program chopping*. Program dicing automatically identifies a set of statements which are likely to contain the bug by using the information that some variables fail some tests while other variables pass all tests. a program dicing is obtained using set operations on one or more backward program slices. Slices can be combined with each other in different ways: for eaxmple, the intersection of two slices contains all statements that lead to an error in both test cases; the intersection of slice A with the complement of slice B excludes from slice A all statements that do not lead to an error in the second test case.

The original work on dicing considered only backward slices. Incorporating forward slices increases the usefulness of dicing. For example, program chopping identifies statements statements that lie between two points a and b in the program which will be affected by a change made at a. A program chop is useful in debugging when a change at a causes an incorrect result to be produced at b. Debugging should be focussed on the statements between a and b that transmit the change of a to b.

2.4.3 Testing

Software maintainers are often faced with regression testing. Regression testing is the task of retesting software after a modification. This process may involve running the modified program on a large number of test cases, even after the smallest of changes. Although the effort required to make a small change may be minimal, the effort required to retest a program after such a change may be substantial. While decomposition slicing eliminates the need for regression testing on the complement, there still may be a substantial number of tests to be run on the dependent, independent and changed parts. Slicing can be used to reduce the number of these tests.

2.4.4 Software Maintenance

Software maintenance is a costly process because it is a very much tedious task to understand existing software and make changes without having a negative impact on the unchanged part. One of the problems in software maintenance is that of the *ripple effect*, i.e., whether a change in a code of the program will affect the behavior of other codes of the program. This problem can be avoided by knowing which variables in which statements will be affected by a modified variable and which variables in which statements will affect a modified variable during software maintenance.

A new kind of slice, called a *decomposition slice*, is useful in making a change to a piece of software without introducing any bugs. A decomposition slice captures all computations of a variable and is independent of program location. It is useful to a maintainer.

2.4.5 Program Integration

Programmers often face the problem of integrating several related, but slightly different, variants of a system. *Program integration* is concerned with the process of merging multiple variants of a program's source code. Let *Base* be a program and A and B be its two variants, each created by modifying separate copies of *Base*. Then, the goal of program integration is to determine the interferance of the modifications and to create an integrated program that incorporates both sets of changes as well as the portions of *Base* preserved in both variants in case there is no interferance.

The first step in program integration is to look for textual differences. More sophisticated techniques can be found in literature [17, 58]. *Semantic-based* program integration is a technique that creates an integrated program incorporating the changed computations of the variants and the computations of the base program that are preserved in all variants. Horwitz et al. [58] developed an algorithm for semantic-based program integration that creates an

integrated program by merging certain program slices of the variants. Their algorithm takes as input three programs, *Base* and its two variants A and B. The integrated program is produced by

1. constructing program dependence graphs for *Base*, A and B
2. determining the differences in behavior of a variant
3. merging the program dependence graphs by combining the program slices of *Base*, A and B.
4. testing the two versions for interferance
5. reconstructing a program from the merged program dependence graph

2.4.6 Software Quality Assurance

Software quality assurance auditors are faced with the task of locating safety critical code that may be interleaved throughout the entire system and ascertaining its effects throughout the system. Program slicing can be used to locate all code that contributes to the value of variables that might be part of a safety critical component.

Slicing-based techniques can be used to validate functional diversity i.e. there are no interactions of one safety critical component with another safety critical component and there are no interactions of non-safety critical component with the safety critical components. If two output values are critical, then these output values should be computed independently. They should not depend on the same internal functions, since the same error might manifest in both output values in the same way and the error might be hidden. The use of functional diversity becomes useful to defend against such errors. Functional diversity allows the same function to be executed along two or more independent paths. The critical output values depend on different internal functions. Program slicing can be used to determine the logical independence of the slices computed for the two output values.

2.4.7 Functional Cohesion

Cohesion is an attribute of a software unit that purports to measure the "relatedness" of the unit. Cohesion has been qualitatively characterized as *coincidental*, *logical*, *procedural*, *communicational*, *sequential* and *functional*, with coincidental being the weakest and functional being the strongest.

Biemann and Ott [38] define *data slices* to construct a slicing-based measure of functional cohesion. Data slice is a backward and forward static slice that uses data tokens instead

of statements as the unit of decomposition. Data tokens may be variables and constant definitions and references. The tokens that are common to more than one data slice are the connections between the slices. They are referred to as *glue*. The *glue* binds the slices together. The tokens that are in every data slice of a function are known as *super-glue*. *Strong Functional Cohesion* is measured as the ratio of *super-glue* tokens to the total number of tokens in the slice. *Weak Functional Cohesion* is expressed as the ratio of *glue* tokens in the slice to the total number of tokens in the slice.

Another method for measuring cohesion is to measure the *adhesiveness* of the individual tokens. The *adhesion* of an individual token is the ratio of number of slices in which the token appears to total the number of data slices in a procedure. For example, a token that glues five data slices together is more adhesive than a token that glues only two data slices together.

2.4.8 Reverse Engineering

Reverse engineering concerns the problem of comprehending the current design of a program and the way this design differs from the original design. This involves abstracting out of the source code the design decisions and rationale from the initial development (design recognition) and understanding the algorithms chosen (algorithm recognition).

Program slicing provides a toolset for this type of re-abstraction. For example, a program can be displayed as a lattice of slices ordered by the *is-a-slice-of* relation. Comparing the original lattice and the lattice after (years of) maintenance can guide an engineer towards places where reverse engineering energy should be spent. Because slices are not necessarily contiguous blocks of code they are well suited for identifying differences in algorithms that may span multiple blocks or procedures.

Chapter 3

OVERVIEW OF RELATED WORK

Basic Program Slicing Techniques

Slicing of Object-Oriented Programs

Slicing of Aspect-Oriented Programs

This chapter presents an overview of the basic program slicing techniques and a brief history of their developments. Then, the work done by various researchers on slicing of aspect-oriented programs are discussed.

3.1 Basic Program Slicing Techniques

Ferrante et al. [27] introduced an intermediate program representation, called the *program dependence graph* (PDG), that makes explicit both the data and control dependences for each operation in a program. They developed an algorithm to determine the data dependences and control dependences. Ottenstein and Ottenstein [30] first considered program slicing as a graph reachability problem in a dependence graph. They used program dependence graph (PDG) to find static slice of a program with single procedure. Harman and Danicic [40] introduced the effect minimal slice, which separates the semantic concept of a slice. According to them, an effect minimal slice maintains the effect of the original program upon a chosen set of variables, but it drops the constraint that a slice must be a subset of the original program. Horwitz et al. [57] developed a *system dependence graph* (SDG) as an intermediate program representation for procedural programs with multiple procedures. They proposed a two-phase graph reachability algorithm on the SDG to compute inter-procedural slice. In the first phase, all the edges in the SDG except parameter-out edges are followed backward and the corresponding vertices are marked. In the second phase, all the edges except call edges and parameter-in edges are followed backward from the vertices marked during the first phase. The slice consists of the union of vertices marked in both the phases.

Korel and Yalamanchili [10] developed a forward algorithm to compute forward dynamic program slices. The major advantage of the forward approach is that space complexity is bounded as opposed to the backward methods of slice computation. Korel and Rilling [9] introduced the concept of *call graph* slicing. Call graph slicing is the slicing of a program at the call-graph level. On the call-graph level, a program is represented by a set of modules (procedures) and a set of call relationships between modules, where each module is graphically represented by a rectangle and each call relationship by a line connecting two modules. They developed a slicing tool to support program slicing on the source code level and on the call-graph level.

Kamkar [41] proposed various notions of execution slice and a method for inter-procedural dynamic slicing. Zhang et al. [67] discussed the design and evaluation of three precise dynamic slicing algorithms called the *full preprocessing* (FP), *no preprocessing* (NP) and *limited preprocessing* (LP) algorithms. Their algorithms differ in the relative timing of constructing the dynamic data dependence graph and its traversal for computing requested dynamic slices. Experimentally, they showed the LP algorithm to be a fast and practical precise

algorithm. The concept of *call-mark* slicing was introduced by Nishimatsu et al. [5]. The call-mark slicing combines static analysis of a program's structure with lightweight dynamic analysis. The data dependences and control dependences among the program statements are statically analyzed beforehand and the procedure or function calls are marked during execution. Using this information, the dynamic dependences of the variables are explored.

3.2 Slicing of Object-Oriented Programs

Larsen and Harrold [37] extended the SDG of Horwitz et al. [57] to represent object-oriented programs incorporating many object-oriented features such as classes, objects, inheritance, polymorphism etc. After constructing the SDG, they have used a two-phase algorithm to compute static slice of an object-oriented program. The limitation of this approach is unnecessary computation for calculating data dependences. Also, the case where an object is used as a parameter or as a data member of other objects is not considered. Moreover, the data members for different objects instantiated from the same class are not distinguished and the resulting slice is imprecise. Liang and Harrold [18] developed a more efficient intermediate representation of object-oriented programs which is an extension to the SDG of Larsen and Harrold [37]. Their SDG represents objects that are used as parameters or data members in other objects, the effects of polymorphism on parameters and parameter bindings. The data members for different objects can be distinguished using this approach. Later many researchers have extended the work on static slicing of object-oriented programs [18, 69].

Also dynamic slicing of OOPs have been addressed by several researchers [31, 46, 48, 49]. Korel and Laski [8] introduced the concept of dynamic program slicing. Agarwal and Horgan [25] first proposed the algorithms for finding dynamic slices using dependence graphs.

Zhao [31] extended the *dynamic dependence graph* (DDG) of Agarwal and Horgan [25] for the representation of various dynamic dependencies between statement instances for a particular execution of an object-oriented program. He named this graph as *dynamic object-oriented dependence graph* (DODG). The DODG is an arc-classified digraph (V, A), where V is the multi-set of flow-graph vertices and A is the set of arcs representing dynamic control dependencies and data dependencies between vertices. His construction of DODG is based on dynamic analysis of control flow [24]. The DODG is constructed by creating a new vertex for each occurrence of a statement in the execution history and creating all the dependence edges associated with the occurrence at run-time. The DODG of the example program in Figure 3.1 is shown in Figure 3.2.

Zhao has adopted the following concepts for dynamic slicing of object-oriented programs:

```
    public class TestPrime{
          private static int n;
          private static boolean r;
1:        public static void main(String[] args){
2:              n=Integer.parseInt(args[0]);
3:              r=prime(n);
4:              if(r)
5:                    System.out.println("Prime");
                else
6:                    System.out.println("Not Prime");
          }

7:        public static boolean prime(int n){
                int i;
                boolean k;
8:              i=2;
9:              while(i<n){
10:                   if(n%i==0){
11:                         k=false;
12:                         break;
                      }
                      else
13:                         k=true;
14:                   i++;
                }
15:             return k;
          }
    }
```

Figure 3.1: An example Java program

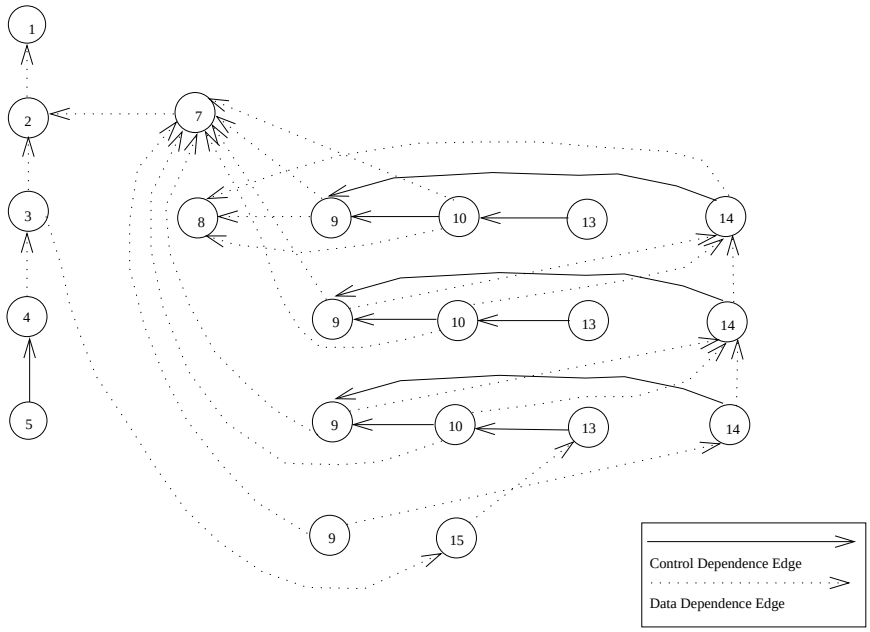

Figure 3.2: The DODG of the program given in Figure 3.1 on input argv[0]=5

- A *dynamic slicing criterion* for an object-oriented program P is of the form (s, v, t, i), where s is a statement in the program, v is a variable used at s and t is an execution trace of the program with input i. The execution trace or length of execution of a program under a given test case is the sequence $< S_1, S_2, \ldots, S_n >$ of statements in the program. They are appended in the order in which they are visited during execution.

- A *dynamic slice* of an object-oriented program P on a given dynamic slicing criterion (s, v, t, i) consists of all statements in the program that actually affected the value of a variable v at statement s for the given input i.

Zhao has used a two-phase algorithm on the DODG for the computation of dynamic slices of object-oriented programs. The two-phases of the algorithm are:

1. Computing a dynamic slice over the DODG of the object-oriented program. This is done by performing either breadth-first or depth-first traversal on the DODG of the program by taking the vertex corresponding to the statement of interest as the starting point.

2. Mapping the slice over the DODG to the source code to obtain a dynamic slice of the program. This is done by simply defining a mapping function.

The drawback of this approach is that the number of nodes may be unbounded for programs having many loops since the number of nodes is equal to the number of executed statements. Also, the space complexity is $O(S)$ and time complexity is $O(S^2)$, where S is the length of execution of the program.

Song et al. [64] proposed a method to compute forward dynamic slices of object-oriented programs using *dynamic object relationship diagram* (DORD). They computed the dynamic slices for each statement immediately after the statement is executed. The dynamic slices of all executed statements have been obtained after the execution of last statement.

Xu et al. [12] extended their earlier work [69] to dynamically slice object-oriented programs. Their method uses *object program dependence graph* (OPDG) and other static information to reduce the information to be traced during execution and computes dynamic slices combining static dependence information and dynamic execution of the program.

Wang et al. [65] proposed a new algorithm for dynamic slicing of Java programs which operates on compressed bytecode traces. According to their approach, first, the bytecode stream corresponding to an execution trace of a Java program is compactly represented. Then, a backward traversal of the compressed program trace is performed to compute data/control dependences on-the-fly. The slice is updated as these dependences are encountered during trace traversal.

Mohapatra et al. [49, 50] have developed edge marking and node marking dynamic slicing technique for object-oriented programs. Many researchers [11, 31, 47, 48, 51, 52] have extended the work of dynamic slicing of object-oriented features. But none of these researchers have considered the aspect-oriented features.

3.3 Slicing of Aspect-Oriented Programs

Zhou et al. [68] proposed a technique to test aspect-oriented software. Their technique selects test cases that are relevant to aspects under test and specify the sufficiency of test cases on the aspect being tested.

Zhao [34] developed data-flow based unit-testing algorithm for aspect-oriented programs. His approach tests two types of units for an aspect-oriented program, i.e., aspects that are modular units of cross-cutting implementation of the program and those classes whose behavior may be affected by one or more aspects. For each aspect or class, this approach performs three levels of testing, i.e., intra-module, inter-module and intra-aspect or intra-class testing. For an individual module such as a piece of advice, a piece of introduction and a method, intra-module testing is performed. For a public module along with other modules it calls in an aspect or class, inter-module testing is performed. For modules that can be accessed outside the aspect or class and can be invoked in any order by users of the aspect or class, intra-aspect or intra-class testing is performed.

Balzarotti et al. [15] proposed an approach to slice AspectJ programs based on the analysis of the woven code. According to their approach, first, the classes and aspects are compiled using the AspectJ compiler and aspects are weaved into the executable program. Then, the existing slicing algorithms are applied to the resulting Java bytecode and slices are obtained. The slices are a set of bytecode statements. Finally, the results are mapped onto original aspect-oriented source code.

Ishio et al. [62] developed a program debugging tool using AspectJ. They [61] also applied aspect-oriented programming technique to calculate program slice. According to their approach, the program dependence graph (PDG) of Ottenstein and Ottenstein [30] is constructed first. Then, the vertices of the PDG are traversed in reverse order from the vertex of interest in order to calculate the slice.

For the first time, Zhao [33] developed the aspect-oriented system dependence graph (ASDG) to represent aspect-oriented programs. The ASDG is constructed by combining the SDG for non-aspect code, the aspect dependence graph (ADG) for aspect code and some additional dependence arcs used to connect the SDG and ADG. Then, he used the two-pass slicing algorithm proposed by Larsen and Harrold [37] to compute static slice of aspect-oriented programs.

Later, Zhao and Rinard [35] developed an algorithm to construct SDG for aspect-oriented programs. Braak [59] extended the ASDG proposed by Zhao [33, 35] to include inter-type declarations in the graph and performed forward slicing to find static slice of an aspect-oriented program. Many developments on aspect-oriented programming have been found in the literature [6, 14, 16, 23, 32, 36, 63].

All the above mentioned work focus on static slicing of aspect-oriented programs. We have not come across any work discussing dynamic slicing of aspect-oriented programs.

Chapter 4

ASPECT-ORIENTED PROGRAMMING

Basic Concepts

AspectJ: An Aspect-Oriented Programming Language

Features of AspectJ

4.1 Basic Concepts

Gregor Kiczales et al. [22] originated the concept of Aspect-Oriented Programming (AOP) at Xerox Palo Alto Research Center (PARC) in 1996. An *aspect* is an *area of concern* that cuts across the structure of a program. *Concern* is defined as some functionality or requirement necessary in a system, which has been implemented in a code structure [3, 6, 23, 62]. Examples of aspects are data storage, user interface, platform-specific code, security, distribution, logging, class structure, threading etc.

The strength of aspect-oriented programming is allowing separation of concerns, by permitting the programmer to create cross-cutting concerns as program modules. *Cross-cutting concerns* are those parts, or aspects, of the program that end up scattered across multiple program modules, and tangled with other modules in standard design.

For instance, let us consider the example program shown in Figure 4.1. The objective of this program is to transfer an amount from one account to another in a banking application. In this example, various cross-cutting concerns such as transactions, security, logging etc. are tangled with the basic functionality (sometimes called the *business logic concern*). If there is a need to change the security considerations for the application, then it would require a major effort since security-related operations appear scattered across numerous methods. This means that the cross-cutting concerns do not get properly encapsulated in their own modules and this increases the system complexity.

```
void transfer(Account fromAccount, Account toAccount, int amount){
  if (!getCurrentUser().canPerform(OP_TRANSFER)){
    throw new SecurityException();
  }
  if (amount<0){
    throw new NegativeTransferException();
  }
  if (fromAccount.getBalance()<amount){
    throw new InsufficientFundsException();
  }
  Transaction tx=database.newTransaction();
  try{
    fromAccount.withdraw(amount);
    toAccount.deposit(amount);
    tx.commit();
    systemLog.logOperation(OP_TRANSFER,fromAccount,toAccount,amount);
  }
  catch(Exception e){
    tx.rollback();
  }
}
```

Figure 4.1: An example program

The goal of aspect-oriented programming (AOP) is to make it possible to deal with cross-cutting aspects of a system's behavior as much in isolation as possible [63]. Although

the inherent modularity of object-oriented languages are extremely useful in this respect, they are unable to modularize cross-cutting cencerns in complex systems. Aspect-Oriented Programming provides specific language mechanisms to explicitly capture the cross-cutting structure.

To better support the expression of cross-cutting design decisions, AOP uses a *component language* to describe the basic functionality of the system and an *aspect language* to describe the diffrent cross-cutting properties. The components and the aspects are then combined into a system using an *aspect weaver* [14]. The *aspect weaver* makes it possible for an advice to be activated at appropriate join points during run-time. Thus, a source code is modifed by inserting aspect-specific statements at join points.

4.2 AspectJ: An Aspect-Oriented Programming Language

Several aspect-oriented programming languages have been prposed such as AML (Aspect Markup Language), an environment for sparse matrix computation [29], RG (Reverse Graphics), an environment for creating image processing systems [4]. A most popular AOP language is AspectJ. Other aspect-oriented frameworks include COOL (COOrdination Language) for expressing synchronization concerns [13], RIDL (Remote Invocation Data transfer Language) for expressing distribution concerns [13], JBOSS, Spring AOP, AspectWerkz [14, 21].

AspectJ, created by Chris Maeda [22] at Xerox Palo Alto Research Center (PARC), is essentialy an aspect-oriented extension to Java programming language. In other words, we can say that AspectJ is compatible with current Java platform [20]. There are four types of compatibility :

- *Upward compatibility* - all legal Java programs must be legal AspectJ programs.
- *Platform compatibility* - all legal AspectJ programs must run on standard Java virtual machines.
- *Tool compatibility* - it must be possible to extend existing tools to support AspectJ in a natural way; this includes IDEs, documentation tools, and design tools.
- *Programmer compatibility* - Programming with AspectJ must feel like a natural extension of programming with Java.

4.3 Features of AspectJ

AspectJ adds some new features to Java. These features include *join points*, *pointcut*, *advice*, *aspect*, *introduction* or *inter-type declaration*.

- *Join Points* - These are well-defined points in the execution of a program, such as method call, method execution and method reception join points (a point where a method received a call, but this method is not executed yet).

- *Pointcut* - This is a means of referring to collections of join points and certain values at those join points. AspectJ defines several primitive *pointcut designators* that can identify all types of join points. For example, in Figure 4.2, the pointcut *factorialOperation* at statement 13 picks out join points i.e., the pointcut *factorialOperation* picks out each call to the method *factorial()* of an instance of the class *TestFactorial*, where an *int* is being passed as an argument and it makes the value of that argument to be available to the enclosing advice or pointcut.

- *Advice* - It is method-like construct used to define additional behavior at join points. This is used to define some code that is executed when a pointcut is reached. Advice brings together a pointcut (to pick out join points) and a body of code (to run at each of those join points). There are three types of advice in AspectJ: *after*, *before*, *around*.

  (i) *After- After advice* on a particular join point runs after the program proceeds with that join point. For example, after advice on a method call join point runs after the method body has run, just before control is returned to the caller.

  (ii) *Before- Before advice* runs as a join point is reached, before the program proceeds with the join point. For example, before advice on a method call join point runs before the actual method starts running, just after the arguments to the method call are evaluated. For example, in Figure 4.2, the *before* advice at statement 14 runs just before the join points picked out by the pointcut *factorialOperation*.

  (iii) *Around- Around advice* on a join point runs as the join point is reached, and has explicit control over whether the program proceeds with the join point.

  Additionally, there are two special cases of after advice: *after returning* and *after throwing*, corresponding to the two ways a sub-computation can return through a join point.

  (i) *After returning- After returning* advice runs just after each join point picked out by the pointcut, but only if it returns normally. The return value can be accessed. After the advice runs, the return value is returned. For example, in Figure 4.2,

the *after returning* advice at statement 16 runs just after each join point picked out by the pointcut *factorialOperation*, but only if it returns normally. The return value can be accessed and it is named *result* in Figure 4.2 at statement 16. After the advice runs, the return value is returned.

(ii) *After throwing*- *After throwing* advice runs just after each join point picked out by the pointcut, but only when it throws an exception. The advice re-raises the exception after it is done.

- *Aspect* - These are units of modular crosscutting implementation composed of pointcuts, advice, and ordinary JAVA member declarations. An *aspect* is a cross-cutting type, defined by the aspect declaration. Aspects are defined by *aspect* declarations, which have a similar form of class declarations. For example, in Figure 4.2, there is one aspect named *OptimizeFactorialAspect* at statement 12.

- *Introduction* or *Inter-Type Declaration* - It allows an aspect to add methods, fields or interfaces to existing classes. It can be *public* or *private*. An introduction declared as *private* can be referred to or accessed *only* by the code in the aspect that declared it. An introduction declared as *public* can be accessed by *any* code.

- *Pointcut Designator* - It is a formula that specifies the set of join points to which a piece of advice is applicable. A pointcut designator identifies all types of join points. A pointcut designator simply matches certain join points at runtime. For example, in Figure 4.2, the pointcut designator

  call (long TestFactorial.factorial(int))

  at statement 13 matches all method calls to *factorial* from an instance of the class *TestFactorial*.

  Pointcuts can be combined using logical operators *and* (&&), *or* (||) and *not* (!). For example, in Figure 4.2, the compound pointcut designator

  call (long TestFactorial.factorial(int)) && args(n)

  at statement 13 refers to all method calls to *factorial()* of an instance of *TestFactorial*, where the argument of type *int* is passed to the method *factorial()*.

  User-defined pointcut designators are defined with *pointcut* declaration. For example, in Figure 4.2, the declaration

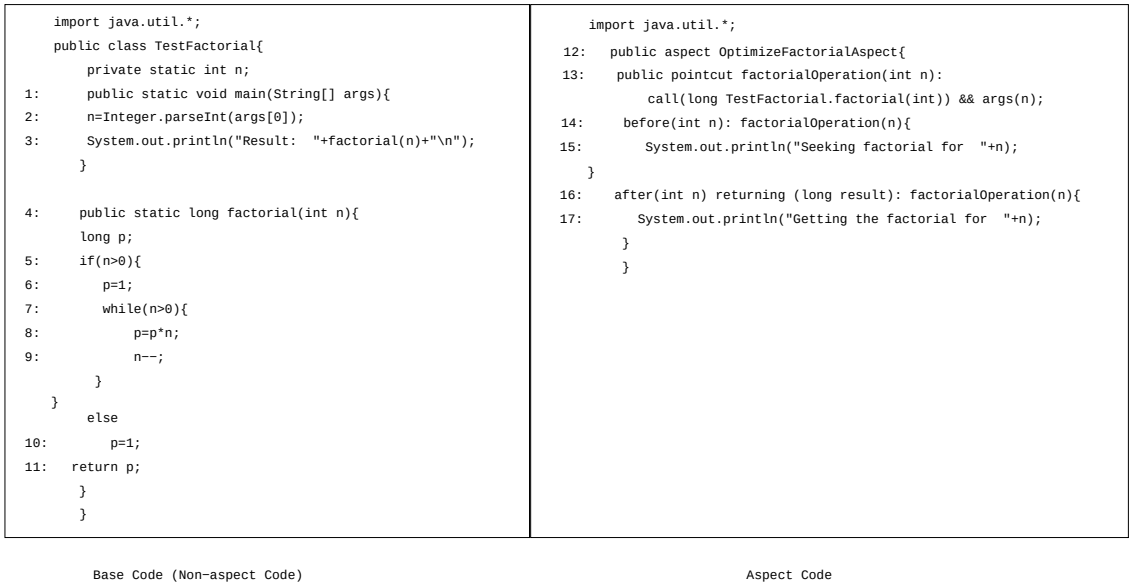

Figure 4.2: An Example AspectJ Program

public pointcut factorialOperation(int n):
call (long TestFactorial.factorial(int)) && args(n)

at statement 13 defines a new pointcut designator, *factorialOperation*, that specifies a call to the method *factorial()* of an instance of *TestFactorial* and the argument passed to the method to be of type *int*.

An AspectJ program is divided into two parts: base code or non-aspect code and aspect code. The *base code* includes classes, interfaces and other standard Java constructs.The *aspect code* implements the cross-cutting concerns in the program. For example, Figure 4.2 shows an AspectJ program for finding the factorial of a number. The program is divided into the base code or non-aspect code containing the class *TestFactorial* and the aspect code *OptimizeFactorialAspect* containing the advices and pointcuts. Any AspectJ implementation ensures that both the codes i.e., aspect code and base code run together in a properly coordinated fashion. Such type of process is called *aspect weaving*. The key component for this process is *aspect-weaver* which makes the applicable advices to run at the appropriate join points.

Chapter 5

COMPUTATION OF DYNAMIC SLICES OF ASPECT-ORIENTED PROGRAMS

Basic Concepts and Definitions

The Dynamic Aspect-Oriented Dependence Graph (DADG)

Computation of Dynamic Slices of Aspect-Oriented Programs

5.1 Basic Concepts and Definitions

In this section, we present the basic concepts and the definitions which will be used in our algorithm.

Definition 1: A *digraph* is an ordered pair (V, A) where V is a finite set of elements called *vertices* and $A \subseteq V \times V$.
Definition 2: An *arc-classified digraph* is an n-tuple $(V, A_1, A_2, \ldots, A_{n-1})$ such that every (V, A_i), $(i = 1, 2, \ldots, n-1)$ is a digraph and $A_i \cap A_j = \phi$ for $i = 1, 2, \ldots, n-1$ and $j = 1, 2, \ldots, n-1$.
Definition 3: The *flow graph* of an aspect-oriented program is a digraph (V, A), where V is the set of vertices corresponding to statements and predicates and A is the set of arcs or edges between vertices in V. An arc from vertex u to v means that control passes from vertex u to vertex v during program execution. A feasible path is an executable path for some input data.
Definition 4: An *execution trace* is a path that has actually been executed for some input data.

For example, for the input data argv[0]=4, the order of execution of the statements of the program given in Figure 4.2 is 1, 2, 3, 13, 14, 15, 4, 5, 6, 7, 8, 9, 7, 8, 9, 7, 8, 9, 7, 8, 9, 7, 16, 17, 11. This *execution trace* is given in Figure 5.1.
Definition 5 (Def(var)): Let *var* be a variable in a class in the program P. A node u of the DADG of P is said to be a *Def(var)* node if u represents a definition (assignment) statement that defines the variable *var*.

In the DADG of Figure 5.2, nodes 6 and 8 are the *Def(p)* nodes.
Definition 6 (DefSet(var)): The set *DefSet(var)* denotes the set of all *Def(var)* nodes.

In the DADG of Figure 5.2, *DefSet(p)*={6, 8}.
Definition 7 (Use(var)): Let *var* be a variable in a class in the program P. A node u of the DADG of P is said to be a *Use(var)* node if u represents a statement that uses the variable *var*.

In the DADG of Figure 5.2, nodes 8 and 11 are the *Use(p)* nodes.
Definition 8 (UseSet(var)): The set *UseSet(var)* denotes the set of all *Use(var)* nodes.

In the DADG of Figure 5.2, *UseSet(p)*={8, 11}.

Dynamic slicing of aspect-oriented programs is similar to that of object-oriented programs. However, due to presence of pointcuts and advices, the tracing of dependences becomes much more complex.

Here, we formally define some notions of dynamic slicing of aspect-oriented programs. Let P be an aspect-oriented program and $G = (V, A)$ be the DADG of P. We compute the

```
          public class TestFactorial
          private static int n;
1(1):     public static void main(String[ ] args)
2(1):     n=Integer.parseInt(args[0]);
3(1):     System.out.println("Result:  "+factorial(n)+"\n");
13(1):    public pointcut factorialOperation(int n): call(long TestFactorial.factorial(int)) && args(n);
14(1):    before(int n): factorialOperation(n)
15(1):    System.out.println("Seeking factorial for  "+n);
4(1):     public static long factorial(int n)
5(1):     if(n>0)
6(1):     p=1;
7(1):     while(n>0)
8(1):     p=p*n;
9(1):     n--;
7(2):     while(n>0)
8(2):     p=p*n;
9(2):     n--;
7(3):     while(n>0)
8(3):     p=p*n;
9(3):     n--;
7(4):     while(n>0)
8(4):     p=p*n;
9(4):     n--;
7(5):     while(n>0)
16(1):    after(int n) returning(long result): factorialOperation(n)
17(1):    System.out.println("Getting the factorial for  "+n);
11(1):    return p;
```

Figure 5.1: Execution trace of the program given in Figure 4.2 for argv[0]=4

dynamic slice of an aspect-oriented program with respect to a slicing criterion.

- A *slicing criterion* for an aspect-oriented program is of the form $< p, q, e, n >$, where p is a statement, q is a variable used at p and e is an execution trace of the program with input n.
- A *dynamic slice* of an aspect-oriented program for a given slicing criterion $< p, q, e, n >$ consists of all the statements that have actually affected the value of the variable q at statement p.

Let DS_G be the dynamic slice of G on a given slicing criterion $< p, q, e, n >$. Then, DS_G is a subset of vertices of G, $DS_G(p, q, e, n) \subseteq V$, such that for any $p' \in V$, $p' \in DS_G(p, q, e, n)$ if and only if there exists a path from p' to p in G.

5.2 The Dynamic Aspect-Oriented Dependence Graph (DADG)

In this section, we describe the definition and construction of the dynamic aspect-oriented dependence graph (DADG).

The DADG is an *arc-classified digraph* (V, A), where V is the set of vertices that correspond to the statements and predicates of the aspect-oriented programs, and A is the set of arcs between vertices in V representing dynamic dependence relationships that exist between statements. In the DADG of an aspect-oriented program, following types of dependence arcs may exist.

- control dependence arc
- data dependence arc
- weaving arc

Control dependences represent the control flow relationships of a program i.e., the control predicates on which a statement or an expression depends during execution.
Data dependences represent the relevant data flow relationships of a program i.e., the flow of data between statements and expressions.
Weaving arc reflects the joining of aspect code and non-aspect code at join points.

For example, in Figure 5.2, there is an weaving arc from vertex 13 to vertex 3 to connect the vertex 13 to vertex 3 at the corresponding join point because, there is a function call at statement 3 and the corresponding pointcut at statement 13 captures that function call. Statement 14 represents a *before* advice. This means that the *advice* is executed before the control goes to the corresponding function. So, we add a *weaving arc* from vertex 4 to vertex 15. Similarly, statement 16 represents a *after* advice. This means that the *advice* is executed after the function has been executed and before the control goes to the calling function. That's why we add a weaving arc from vertex 16 to vertex 7. After the execution of *after* advice at statement 17, the control goes to statement 11 where it returns a value to the calling function. So, a weaving arc is added from vertex 11 to vertex 17. Our construction of dynamic aspect-oriented dependence graph of an aspect-oriented program is based on dynamic analysis of control flow and data flow of the program.

The DADG of the program in Figure 4.2 corresponding to execution trace in Figure 5.1 is given in Figure 5.2. In this figure, circles represent program statements, dotted lines represent data dependence arcs, solid lines represent control dependence arcs and dark dashed lines represent weaving arcs.

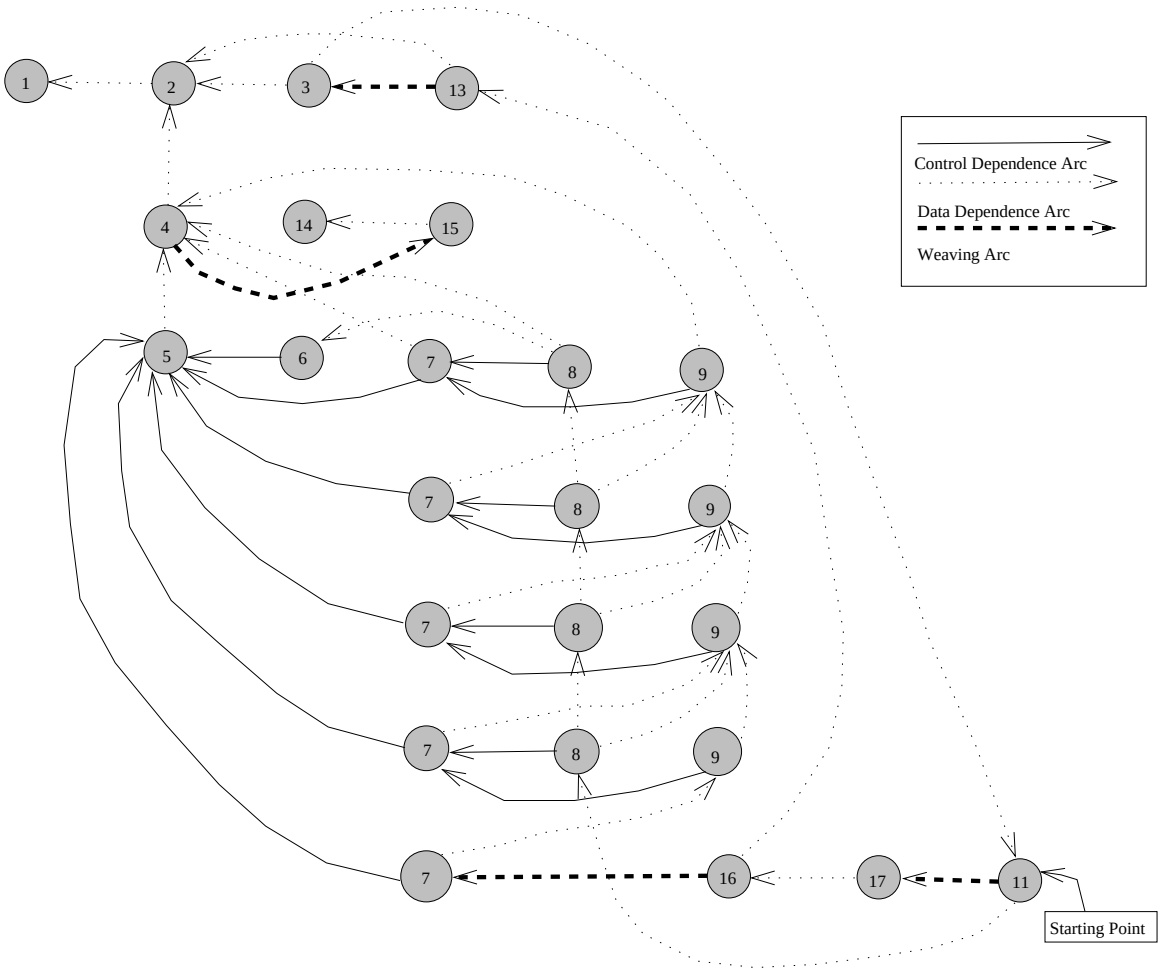

Figure 5.2: Dynamic Aspect-Oriented Dependence Graph for the execution trace given in Figure 5.1

5.3 Computation of Dynamic Slices of Aspect-Oriented Programs

In Section 5.1, we have discussed some notions of dynamic slices. But, this gives only the general views of dynamic slicing of aspect-oriented programs and does not give the procedure for computing them. In this section, we present an algorithm to compute a dynamic slice of an aspect-oriented program. We have named this algorithm trace file based algorithm. Our algorithm first stores the execution history in a trace file called execution trace file. Then, the Dynamic Aspect-oriented Dependence Graph (DADG) is constructed with respect to the given execution trace file. The DADG contains vertices corresponding to the executed statements only. Then, the DADG is traversed using breadth-first or depth-first traversal algorithm taking the vertex of interest as the starting point. The traversed vertices are mapped to the source program to obtain the required dynamic slice.

Algorithm: Trace File Based algorithm

1. Creation of execution trace file: To create an execution trace file, do the following:
   (a) For a given input, execute the program and store each statement s in the trace file after it has been executed.
   (b) If the program contains loops, then store each statement s inside the loop in the trace file after each time it has been executed.
2. Construction of DADG: To construct the DADG of the aspect-oriented program P with respect to the trace file, do the following:
   (a) For each statement s in the trace file, create a vertex in the DADG.
   (b) For each occurrence of a statement s in the trace file, create a separate vertex.
   (c) Add all control dependence edges, data dependence edges and weaving edges to these vertices.
3. Computation of dynamic slice: To compute the dynamic slice over the DADG, do the following:
   (a) Perform the usual breadth-first or depth-first graph traversal over the DADG taking any vertex corresponding to the statement of interest as the starting point of traversal.
4. Mapping of the slice: To obtain a dynamic slice of the aspect-oriented program P, do the following:

(a) Define a mapping function $f : DS_G(p, q, e, n) \rightarrow P$.

(b) Map the slice over the DADG to the source code using f since the slice may contain multiple occurrences of the same vertex.

Working of the Algorithm: We illustrate the working of our algorithm with the help of an example. Consider the example AspectJ program given in Figure 4.2. Now, for the input data argv[0]=4, the program will execute the statements 1, 2, 3, 13, 14, 15, 4, 5, 6, 7, 8, 9, 7, 8, 9, 7, 8, 9, 7, 8, 9, 7, 16, 17, 11 in order. These statements are stored in a trace file. Figure 5.1 shows the corresponding execution trace file. Then, the Dynamic Aspect-Oriented Dependence Graph (DADG) is constructed with respect to this trace file in accordance with the step 2 of the algorithm. Figure 5.2 shows the DADG of the trace file given in Figure 5.1. Since, for the input data argv[0]=4, the statements 8 and 9 are executed four times and statement 7 is executed five times, separate vertices are created for each occurrenec of these statements.

Now, let us suppose that we have to compute the dynamic slice for the slicing criterion $< 11, p >$. Starting from the vertex 11, we can perform either the breadth-first search algorithm or depth-first search algorithm on the DADG. The breadth-first search algorithm yields the vertices 11, 17, 8, 16, 7, 8, 9, 7, 13, 5, 9, 7, 8, 9, 9, 3, 2, 4, 7, 8, 9, 1, 15, 7, 6, 14 and the depth-first search algorithm yields the vertices 11, 8, 9, 9, 9, 4, 15, 14, 2, 1, 7, 5, 7, 7, 8, 8, 8, 6, 7, 17, 16, 13, 3, 7, 9. The traversed vertices are shown as shaded vertices in Figure 5.2. Using the mapping function f, we can find the statements corresponding to these vertices. This gives the required dynamic slice which is shown in rectangular boxes in Figure 5.3.

5.3.1 Correctness Proof

In this section, we sketch the proof of *correctness* of our trace file based algorithm.

Theorem 1 *Trace file based algorithm always finds a correct dynamic slice with respect to a given slicing criterion.*

Proof. The proof is given through mathematical induction. Let P be any given aspect-oriented program for which a dynamic slice is to be computed using trace file based algorithm. According to the principle of mathematical induction, the *Base* and *Hypothesis* are to be set up. The *Base* is set up as follows:
For any set of input values to the program, the dynamic slice with respect to a single executed statement is certainly *correct*, according to the definition.
Then, the *Hypothesis* is set up as follows:
During the traversal of the DADG, assume that the algorithm has produced the *correct*

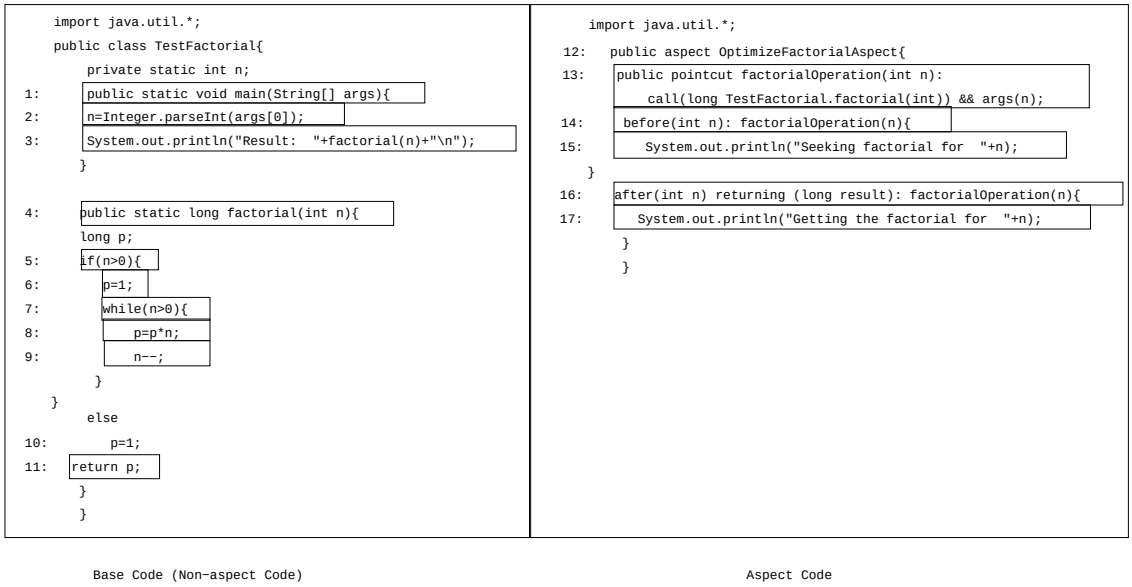

Figure 5.3: The dynamic slice of the program given in Figure 4.2 for the slicing criterion (11,p)

dynamic slices prior to the traversal of the current vertex v.

To complete the proof, we need only to show that the dynamic slice computed after the traversal of the vertex v is also correct. Let u be the vertex traversed prior to the vertex v. This means that there is an edge from u to v in the DADG. The presence of the edge (u, v) signifies that the vertex u has been executed after the execution of v since DADG contains vertices corresponding to the executed statements only and u is dependent (data or control or weaving) on v. Since v is adjacent to u, v can be included in the set of traversed vertices. The set of traversed vertices give the required dynamic slice. Since the dynamic slice after the traversal of u is correct and u is dependent on v, the dynamic slice after the traversal of v must also be correct. This establishes the correctness of the algorithm.

5.3.2 Complexity Analysis

In the following we discuss the space and time complexity of our DADG algorithm.

Space Complexity: Let P be an aspect-oriented program and S be the length of execution of P. Each executed statement will be represented by a single vertex in the DADG. Thus, it can be stated that there are S number of vertices in the DADG corresponding to all executed statements of program P. Also, S numbers of statements are stored in the execution trace file. So, the space complexity of the trace file based algorithm is $O(S)$.

Time Complexity: Let P be an aspect-oriented program and S be the length of execu-

tion of P. The total time complexity is due to four components:

1. time required to store each executed statement in a trace file which is $O(S)$.
2. time required to construct the DADG with respect to the execution trace file which is $O(S)$.
3. time required to traverse the DADG and to reach at the specified vertex which is $O(S^2)$.
4. time required to map the traversed vertices to source program P which is $O(S)$.

So, the time complexity of the trace file based algorithm is $O(S^2)$.

Chapter 6

IMPLEMENTATION

Overview of DDST

Implementation of the Slicing Tool

Experimental Results

In this chapter, we briefly describe the implementation of our algorithm. We have named our dynamic slicing tool *dynamic dependence slicing tool* (DDST) for aspect-oriented programs. First, we present an overview of our slicing tool and then we discuss briefly the implementation of the slicing tool.

6.1 Overview of DDST

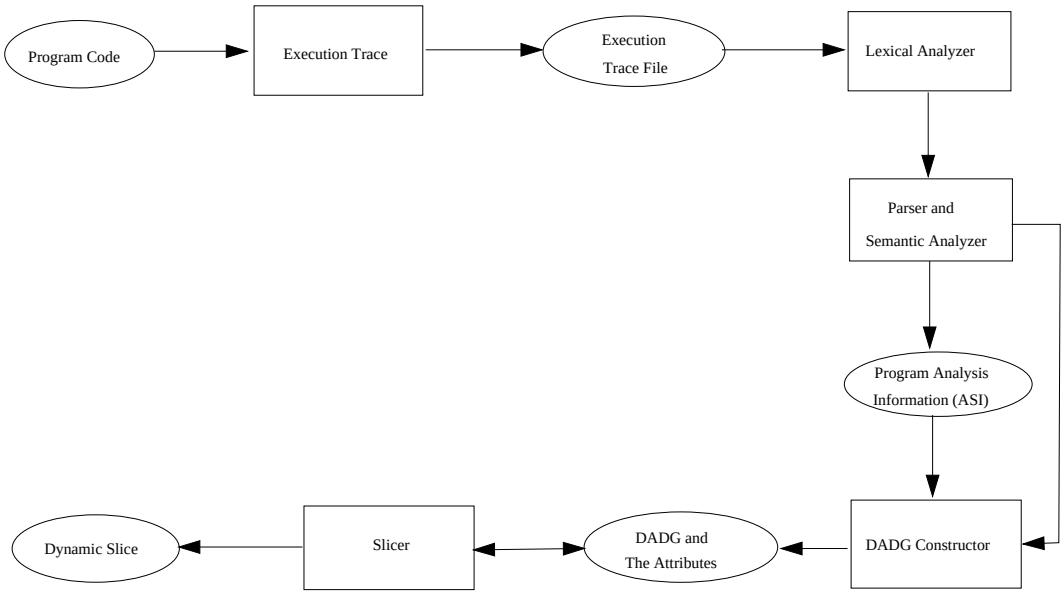

Figure 6.1: Schematic diagram of the slicing tool

The working of the slicing tool is schematically shown in Figure 6.1. The arrows in the figure show the data-flow among the different blocks of the tool. The blocks shown in rectangular boxes represent executable components and the blocks shown in ellipses represent passive components of the slicing tool.

A program written in AspectJ is given as input to DDST. The overall control for the slicer is done through a *coordinator* with the help of a *graphical user interface* (GUI). The *coordinator* takes user input from the GUI, interacts with other relevant components to extract the desired results and returns the output back to the GUI.

The *execution trace* component creates an *execution trace file* for a particular execution of the program. This component takes the user input from the coordinator, stores each executed statement for that input in a file and outputs that file back to the coordinator. This file is called *execution trace file*.

The *lexical analyzer* component reads the execution trace file and breaks it into tokens for the grammar expressed in the parser. When the lexical analyzer component encounters

a useful token in the program, it returns the token to the parser describing the type of encountered token.

The *parser and semantic analyzer* component functions as a state machine. The parser takes the token given by the lexical analyzer and examines it using the grammatical rules laid for the input programs. The semantic analyzer component captures the following important information of the program.

- For each vertex u of the program
  - the lexical successor and predecessor vertices of u,
  - the sets of variables defined and used at vertex u,
  - the type of the vertex: assignment or test or method call or return etc.

The lexical component, parser and semantic analyzer component provide the necessary program analysis information to the *DADG constructor* component. The *DADG constructor* component first constructs the CFG and the post-dominator tree of the program using the basic information provided by the lexical and semantic analyzer components. The inter-statement control dependencies are captured using the CFG and the post dominator tree. Then, it constructs the DADG of the program with respect to the trace file along with all the required information to compute slices and stores it in appropriate data structures.

The *slicer* component traverses the DADG. It takes the user input from the *coordinator* and outputs the computed information back to the *coordinator*. The graphical user interface (GUI) functions as a front end to the slicing tool.

6.2 Implementation of the Slicing Tool

We have implemented our algorithm in C++. We have used the compiler writing tool *Lex and YACC* [55] for *Lexical Analyzer*, *Parser* and *Semantic Analyzer* components of our slicer. Lex is officially known as a "Lexical Analyser". Its main job is to break up an input stream into more usable elements or in other words, to identify the "interesting bits" in a text file. Yacc is officially known as a "parser". Its job is to analyse the structure of the input stream, and operate of the "big picture". In the course of its normal work, the parser also verifies that the input is syntactically sound. YACC stands for "Yet Another Compiler Compiler". This is because this kind of analysis of text files is normally associated with writing compilers. Lex and YACC programs allow one to parse complex languages with ease. The program Lex generates a so called Lexer. This is a function that takes a stream of characters as its input, and whenever it sees a group of characters that match a key, takes a certain action. YACC can parse input streams consisting of tokens with certain values. This

Table 6.1: Encoding used for different types of edges of DADG

Code	Edge Type
0	No Edge
1	Control Dependence Edge (True Case)
2	Control Dependence Edge (False Case)
3	Data Dependence Edge (Loop Independent Case)
4	Data Dependence Edge (Loop Carried Case)
5	Weaving Edge

clearly describes the relation YACC has with Lex, YACC has no idea what input streams are, it needs preprocessed tokens. While one can write his own Tokenizer, that will be left entirely up to Lex.

The sample AspectJ program is executed for a given input. The executed statements are stored in a trace file. This trace file is given as input to the Lex and YACC program. The Lex extracts program tokens and stores the data in a data structure called *statement_info*. The DADG of the AspectJ program is automatically constructed by taking input from the *parser* and *semantic analyzer component* i.e., from YACC. For constructing the DADG, we have used many flags such as *if_flag* to check whether the statement is an *if* statement or not, *while_flag* to check whether the statement is a *while* statement or not etc.

We use an adjacency matrix *dadg[][]* to store the DADG of the given AspectJ program P. This matrix is of the following type:

```
struct edge {
int exist, type;
} edge;
```

- The attribute *exist* has value 0 or 1. `dadg[i][j].exist` is 1 if there is an edge between node number i and j, otherwise 0.
- The data member *type* specifies the type of the edge. The code used for this is as given in Table 6.1.

We store the following additional information along with the DADG:

- The set *Def(var)* for each variable *var* in the aspect-oriented program P.
- The set *Use(var)* for each variable *var* in the aspect-oriented program P.

The sets *Def(var)* and *Use(var)* are stored using arrays.

The additional data structures used for the DDST slicer are given below.

The data structure *statement_info* stores the following information:
type of statement, used variable, defined variable and *statement number.*

This data structure stores the details of statements of the input porgram.

```
public class statement_info {
public:
int type; // type of the statement
var_struct def; // name of the variable defined
var_struct use[]; // name of the variable used
int clno,mno,ano; // the number of the class, method, aspect to which the statement
belongs to
int eno; // no.  of times a statement has been executed
};
```

The data structure *class_info* contains the details of a class, methods and variables defined in that class.

```
public class class_info {
public:
char cl_name[]; // name of the class
method_struct method_store[]; // method details
var_struct var_store[]; // member variables
int mno,vno; // number of methods and variables
};
```

The data structure *var_struct* stores the details of a variable such as its name and where it is defined.

```
public class var_struct {
public:
char var_name[]; // name of the variable
int var_at; // vetex no at which the variable is defined
};
```

The data structure *aspect_info* stores the details of an aspect, pointcuts, advices, introductions and methods defined in that aspect.

```
public class aspect_info {
public:
char asp_name[]; // name of the aspect
method_struct method_store[]; // method details
pointcut_info pc[]; // pointcut details
advice_info adv[]; // advice details
intro_info intr[]; // introduction details
int mno,advno,pcno,intrno; // number of methods, advices, pointcuts and introductions
};
```

The data structure *method_struct* contains the name of the method and the variables defined in the method.

```
public class method_struct {
public:
char method_name[]; // name of the method
int v_no; // number of variables defined
var_struct var_store[]; // variables details
};
```

The data structure *advice_info* stores details of the advice such as its name, its type, pointcut to which it belongs.

```
public class advice_info {
public:
char adv_name[]; // name of the advice
int typeadv; // type of advice
char pc_name[]; // pointcut to which advice belongs
};
```

The data structure *intro_info* stores details of the introduction. It contains the name of the introduction, class to which the introduction belongs etc.

```
public class intro_info {
```

```
public:
char intr_name[]; // name of the introduction
char cl_name[]; // name of the class to which the introduction belongs
int type; // 1 for variable, 2 for method
};
```

The data structure *pointcut_info* stores details of the pointcut such as its name, variables details.

```
public class pointcut_info {
public:
char pc_name[]; // name of the pointcut
char method_name[][]; // name of the methods used in the pointcut
var_struct var_store[]; // variables details
int mno; // number of methods
};
```

The input program is invoked for execution. Each executed statement is stored in an execution trace file. Then, *construct_dadg()* method is invoked to construct the DADG with respect to the execution trace file. To traverse the DADG, the method *traverse_dadg()* is used. This method stores the visited vertices in a one-dimensional integer array: `int visit[]`. The *map()* method is used to map the vertices in `visit[]` to the corresponding program statements. The mapped statement numbers are stored in an integer array: `int mapped[]`. The *compute_slice()* method is invoked to compute the dynamic slice for a given slicing criterion. It uses the *traverse_dadg()* method and the *map()* method for this purpose. The dynamic slice is stored in `mapped[]` and it is displayed through GUI.

6.3 Experimental Results

With different slicing criteria, the algorithm has been tested on many programs for 40-50 runs. The sample programs contain loops and conditional statements. Table 6.2 summarizes the *average run-time requirements* of the trace file based algorithm for several programs. Since we have computed the dynamic slices at different statements of a program, we have calculated the average run-time requirements of our trace file based algorithm. The program sizes are small since right now the tool accepts only a subset of AspectJ constructs. However, the results indicate the overall trend of the performance of the trace file based algorithm.

The results in the table 6.2 indicates that the run-time requirement increases rapidly.

Table 6.2: Average Runtime

Sl No.	Prg. Size (# stmts)	trace file based Algorithm (in Sec.)
1	17	0.11
2	43	0.71
3	69	0.89
4	97	1.07
5	123	1.36
6	245	2.46
7	387	3.96
8	562	5.52

This is due to the fact that separate vertices are created in the DADG during run-time for different executions of the same statement. This is followed by a depth-first or breadth-first graph traversal on DADG to compute the dynamic slice. Thus, average run-time requirement becomes high since considerable time is required to perform the traversal on DADG. Furthermore, the algorithm uses a trace file to store the execution history. The time required to read the data from a trace file is significant and is added to the average run-time while computing dynamic slice. All these result in the increase of average run-time requirement rapidly.

Also, the slice extraction time increases rapidly with the program size. This is due to the reason that the slice is computed only after the whole program is executed, by performing a graph traversal on the DADG.

Chapter 7

CONCLUSION

Contributions

Future Work

The development of efficient dynamic slicing algorithms for aspect-oriented programs was the main aim of our work. In the following, we summarize the important contributions of our work. At the end, some suggestions for future work are given.

7.1 Contributions

In this section, we summarize the important contributions of our work i.e. contributions of development of *Computation of Dynamic Slices of Aspect-Oriented Programs*.

7.1.1 Computation of Dynamic Slices of Aspect-Oriented Programs

First, we have stored the executed statements in an execution trace file. Next, we have constructed a dependence-based intermediate representation for aspect-oriented programs. We have named this representation *Dynamic Aspect-Oriented Dependence Graph* (DADG). The DADG is an arc-classified digraph which represents various dynamic dependences between the statements of an aspect-oriented program for a particular execution. Then, we have developed an algorithm to compute dynamic slices of AOPs using the DADG. Taking any vertex as the starting point, our algorithm performs a graph traversal on the DADG using breadth-first graph traversal or depth-first graph traversal. Then, the traversed vertices are mapped to the original program. The space complexity of our algorithm is shown to be $O(S)$, where S is the length of execution of the program. The time complexity of the algorithm is $O(S^2)$, where S is the length of execution of the program. Moreover, we have proved that our algorithm computes *correct* dynamic slices for any slicing criterion. We have implemented the algorithm to prove its *correctness* experimentally.

7.1.2 Implementation

We have implemented our proposed algorithm i.e. trace file based algorithm to verify its *correctness* experimentally. The slicer has been tested on a large of input programs with several executions and slicing criteria. It has been observed that the slicer computed *correct* dynamic slices for all input slicing criteria. Experimentally, this one validated the *correctness* of our proposed algorithm.

7.2 Future Work

In the following, the possible extensions to our work have been outlined briefly.

- Here, we used an *execution trace file*. The use of this file increases the average run-time. This may be reduced if the algorithm can be modified so that it will not use the execution trace file.

- In our work, we did not consider composite data types such as *arrays* while computing dynamic slices of aspect-oriented programs. Our work can be extended to handle composite data types by developing a suitable frame work.

- In our work, we did not consider dynamic slicing of unstructured programs. Our work can be extended to handle unstructured programs by developing a suitable frame work.

- The work can easily be extended to compute dynamic slices of programs written in other aspect-oriented languages such as AspectWerkz, RIDL, RG etc.

- The slicer can be used to develop efficient debuggers and test drivers for large scale aspect-oriented programs.

- The algorithm can be extended to compute *conditioned slices* with respect to a given condition.

Bibliography

[1] Aspect-Oriented Programming. www.wikipedia.org.

[2] AspectJ. www.eclipse.org/aspectj.

[3] Introduction to AOP. www.media.wiley.com.

[4] Mendhekar A., Kiczales G., and Lamping J. RG: A Case-Study for Aspect-Oriented Programming. Technical report, Xerox Palo Alto Research Center, February 1997.

[5] Nishimatsu A., Jihira M., Kusumoto S., and Inoue K. Call-Mark Slicing: An Efficient and Economical Way of Reducing Slice. In *Proceedings of the 21st International Conference on Software Engineering*, pages 422–431, May 1999.

[6] Dufour B., Goard C., Hendren L., Verbrugge C., Moor O. D., and Sittampalan G. Measuring the Dynamic Behaviour of AspectJ Programs. Technical report, McGill University, School of Computer Science, Sable Research Group and Oxford University, Computing Laboratory, Programming Tools Group, March 2004.

[7] Korel B. Computation of Dynamic Program Slices for Unstructured Programs. *IEEE Transactions on Software Engineering*, 23(1), 1997.

[8] Korel B. and Laski J. Dynamic Program Slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[9] Korel B. and Rilling J. Program Slicing in Understanding of Large Programs. In *Proceedings of 6th International Workshop on Program Comprehension*, pages 145–152, 1998.

[10] Korel B. and Yalamanchili S. Forward Computation of Dynamic Program Slices. In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 66–79, August 1994.

[11] Mund G. B., Mall R., and Sarkar S. Computation of Interprocedural Dynamic Program Slices. *Journal of Information and Software Technology*, 45(8):499–512, June 2003.

[12] Xu B. and Chen Z. Dynamic Slicing Object-Oriented Programs for Debugging'. In *Proceedings of 2nd IEEE International Workshop on Source Code Analysis amd Manipulation (SCAM'02)*, pages 115–122, 2002.

[13] Gianpaolo C., Carlo G., and Mattia M. Coding Different Design Paradigms for Distributed Applications with Aspect-Oriented Programming. In *Proceedings of Workshop on System Distribution: Algorithm, Architecture and Language, WSDAAL*. Italy, 1999.

[14] Murphy G. C., Walker R. J., and Baniassad E. L. A. Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect-Oriented Programming. *IEEE Transactions on Software Engineering*, 25(4):438–455, July-Aug 1999.

[15] Balzarotti D., Ursi A. C. D., Cavallaro L., and Monga M. Slicing AspectJ Woven Code. In *Proceedings of the Foundations of Aspect-Oriented Languages Workshop (FAOL'05)*, Chicago, IL(USA), 2005.

[16] Balzarotti D. and Monga M. Using Program Slicing to Analyze Aspect Oriented Composition. In *Proceedings of Foundations of Aspect-Oriented Languages Workshop at Aspect-Oriented Software Development (AOSD'04)*, Lancaster, UK, 2004.

[17] Binkley D., Horwitz S., and Reps T. Program Integration for Languages Procedure Calls. *ACM Transaction on Software Engineering and Methodology*, 4(1):3–35, January 1995.

[18] Liang D. and Harrold M. J. Slicing Objects Using System Dependence Graph. In *Proceedings of the International Conference on Software Maintenance, IEEE*, pages 358–367, November 1998.

[19] Serini D. and Moor O. D. Static Analysis of Aspects. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, March 2003.

[20] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., and Griswold W. G. An Overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*. Budapest,Hungary, 18-22 June 2001.

[21] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., and Grisworld W. G. Report on An Overview of AspectJ. Notes by Tai Hu, CMSC631 Fall 2002.

[22] Kiczales G., Irwin J., Lamping J., Loingtier J. M., Lopes C. V., Maeda C., and Mendhekar A. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag.

[23] Kiczales G. and Mezini M. Aspect-Oriented Programming and Modular Reasoning. In *Proceedings of the 27th International Conference on Software Engineering, ICSE'05*, May 2005.

[24] Agarwal H., DeMillo R. A., and Spafford E. H. Dynamic Slicing in the Presence of Unconstrained Pointers. In *Proceedings of the ACM Fourth Symposium on Testing, Analysis and Verification (TAV)*, pages 60–73, 1991.

[25] Agarwal H. and Horgan J. R. Dynamic Program Slicing. In *ACM SIGPLAN Notices, Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implememntation PLDI'90*, volume 25 of *6*, pages 246–256, June 1990.

[26] Lorenz D. H. Visitor Beans: An Aspect-Oriented Pattern. In *Workshop Reader of the European Conference on Object-Oriented Programming (ECOOP98)*, Finland, 1998. Springer-Verlag.

[27] Ferrante J., Ottenstein K. J., and Warren J. D. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, July 1987.

[28] Hauck F. J., Becker U., Geier M., Meier E., Rastofer U., and Steckermeier M. AspectIX: A Middleware for Aspect-Oriented Programming. In *Workshop Reader of the European Conference on Object-Oriented Programming (ECOOP98)*, Finland, 1998. Springer-Verlag.

[29] Irwin J., Loingtier J. M., Gilbert J. R., Kiczales G., Lamping J., Mendhekar A., and Shpeisman T. Aspect-Oriented Programming of Sparse Matrix Code. In *Proceedings of International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE), Marina del Rey, CA*. Springer-Verlag, December 1997.

[30] Ottenstein K. J. and Ottenstein L. M. The Program Dependence Graph in a Software Development Environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, volume 19, pages 177–184. ACM SIGPLAN Notices, 1984.

[31] Zhao J. Dynamic Slicing of Object-Oriented Programs. Technical report, Information Processing Society of Japan, May 1998.

[32] Zhao J. Change Impact Analysis for Aspect-Oriented Software Evolution. In *Proceedings of the 5th International Workshop on Principles of Software Evolution*, pages 108–112. Orlando, Florida, USA, ACM Press, May 2002.

[33] Zhao J. Slicing Aspect-Oriented Software. In *Proceedings of 10th International Workshop on Program Comprehension*, pages 251–260, June 2002.

[34] Zhao J. Data-Flow Based Unit Testing of Aspect-Oriented Programs. In *Proceedings of 27th Annual International Conference, Computer Software and Applications Conference*, pages 188–197, November 2003.

[35] Zhao J. and Rinard M. System Dependence Graph Construction for Aspect-Oriented Programs. Technical report, Laboratory for Computer Science, Massachusetts Institute of Technology, USA, March 2003.

[36] Blair L. and Monga M. Reasoning on AspectJ Programmes. In *Proceedings of Workshop on Aspect-Oriented Software Development*, pages 45–50, Essen, Germany, March 2003. German Informatics Society.

[37] Larsen L. and Harrold M. J. Slicing Object-Oriented Software. In *Proceedings of 18th International Conference on Software Engineering*, pages 495–505, March 1996.

[38] Bieman J. M. and Ott L. M. Measuring Functional Cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, August 1994.

[39] Harman M., Binkley D., and Danicic S. Amorphous Program Slicing. *Journal of Systems and Software*, 68(1):45–64, 15 October 2003.

[40] Harman M. and Danicic S. A New Approach to Program Slicing. In *Proceedings of 7th International Software Quality Week*, pages 1–14. San Francisco, May 1994.

[41] Kamkar M. An Overview and Comparative Classification of Program Slicing Techniques. *Journal of Systems and Software*, 31(3):197–214, December 1996.

[42] Storzer M. and Krinke J. Interference Analysis for AspectJ. In *Proceedings of Workshop on Foundations of Aspect-Oriented Languages (FAOL)*, Boston, 2003.

[43] Weiser M. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, March 1981.

[44] Weiser M. Programmers Use Slices When Debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[45] Lunau C. P. Is Composition of Metaobjects = Aspect Oriented Programming. In *Workshop Reader of the European Conference on Object-Oriented Programming (ECOOP98)*, Finland, 1998. Springer-Verlag.

[46] Mohapatra D. P. *Dynamic Slicing of Object-Oriented Programs.* PhD thesis, Indian Institute of Technology, Kharagpur, May 2005.

[47] Mohapatra D. P., Kumar R., Mall R., Kumar D. S., and Bhasin M. Distributed Dynamic Slicing of Java Programs. *Journal of Systems and Software*, 79(12):1661–1678, December 2006.

[48] Mohapatra D. P., Mall R., and Kumar R. Dynamic Slicing of Concurrent Object-Oriented Programs. In *Proceedings of International Conference on Information Technology: Progresses and Challenges (ITPC)*, pages 283–290. Kathamandu, May 2003.

[49] Mohapatra D. P., Mall R., and Kumar R. An Edge Marking Technique for Dynamic Slicing of Object-Oriented Programs. In *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, 2004.

[50] Mohapatra D. P., Mall R., and Kumar R. A Node-Marking Technique for Dynamic Slicing of Object-Oriented Programs. In *Proceedings of Conference on Software Design and Architecture (SODA'04)*, 2004.

[51] Mohapatra D. P., Mall R., and Kumar R. Computing Dynamic Slices of Concurrent Object-Oriented Programs. *Information and Software Technology*, 47(12):805–817, September 2005.

[52] Mohapatra D. P., Mall R., and Kumar R. An Overview of Slicing Techniques for Object-Oriented Programs. Technical report, Informatica, 2006.

[53] Gupta R., Soffa M. L., and Howard J. Hybrid Slicing: Integrating Dynamic Information with Static Analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(4):370–397, October 1997.

[54] Gupta R. and Sofia M. L. Hybrid Slicing: An Approach for Refining Static Slices Using Dynamic Information. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering SIGSOFT'95*, volume 20, pages 29–40. ACM SIGSOFT Software Engineering Notes, October 1995.

[55] Levine J. R., Mason T., and Brown D. *Lex and Yacc.* O'REILLY, 3rd edition, 2002.

[56] Clarke S. and Murphy J. Developing a Tool to Support the Application of Aspect-Oriented Programming Principles to the Design Phase. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, April 1998. IEEE Computer Society.

[57] Horowitz S., Reps T., and Binkley D. Inter-Procedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[58] Horwitz S., Prins J., and Reps T. Integrating Noninterfering Versions of Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[59] Braak T. T. Extending Program Slicing in Aspect-Oriented Programming with Inter-Type Declarations. 5th TSConIT Program, June 2006.

[60] Frank T. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[61] Ishio T., Kusumoto S., and Inoue K. Application of Aspect-Oriented Programming to Calculation of Program Slice. Technical report, ICSE, 2003.

[62] Ishio T., Kusumoto S., and Inoue K. Program Slicing Tool for Effective Software Evolution Using Aspect-Oriented Technique. In *Proceedings of Sixth International Workshop on Principles of Software Evolution*, pages 3–12. IEEE Press, September 2003.

[63] Ishio T., Kusumoto S., and Inoue K. Debugging Support for Aspect-Oriented Program Based on Program Slicing and Call Graph. In *Proceedings of 20th IEEE International Conference on Software Maintenance*, pages 178–187, September 2004.

[64] Song Y. T. and Huynh D. T. Forward Dynamic Object-Oriented Program Slicing. In *Proceedings of IEEE Symposium on Application Specific Systems and Software Engineering and Technology (ASSET'99)*, pages 230–237, Richardson, TX, USA, 03/24/1999-03/27/1999 199.

[65] Wang T. and Roychoudhury A. Using Compressed Bytecode Traces for Slicing Java Programs. In *Proceedings of 26th International Conference on Software Engineering (ICSE'04)*, pages 512–521, 23-28 May 2004.

[66] Binkley D. W. and Gallagher K. B. Program Slicing. *Advances in Computers*, 43, 1996. Academic Press, San Diego, CA.

[67] Zhang X., Gupta R., and Zhang Y. Precise Dynamic Slicing Algorithms. In *Proceedings of 25th International Conference on Software Engineering*, pages 319–329, 3-10 May 2003.

[68] Zhou Y., Richardson D., and Ziv H. Towards a Practical Approach to Test Aspect-Oriented Software. In *Proceedings of SOQUA 2004 (First International Workshop on*

Software Quality) and TECOS 2004 (Workshop Testing Component-Based Systems), pages 1–16, 2004.

[69] Chen Z. and Xu B. Slicing Object-Oriented Java Programs. *ACM SIGPLAN Notices*, 36(4), April 2001.